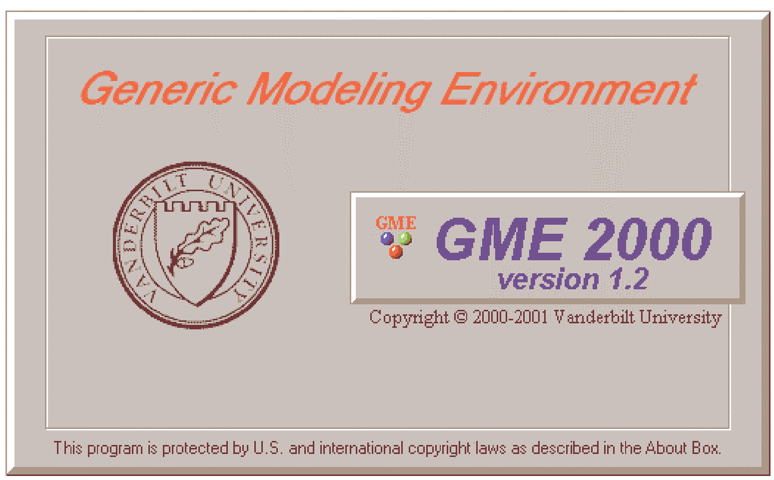
Generic Modeling Environment

GME 2000 User's Manual

Version 1.2
Release 08-15-1
August 2001

Institute for Software Integrated Systems
Vanderbilt University



http://www.isis.vanderbilt.edu

This manual was produced using *Doc-To-Help®*, by WexTech Systems, Inc.

Contents

What's new in version 1.2

User-defined drawing capability

How GME 2000 displays model objects is now decided by external components called decorators. The previous appearance of boxes for models and icons for other objects is preserved as the default visualization (also implemented by decorators included with the GME release). However, users can write their own decorators. The only requirement is that decorators have to implement a COM interface that GME uses when it needs to display the objects. The new UML class diagram paradigm sample comes with its own decorator that displays classnames, stereotypes and attributes inside the class icon and resizes it accordingly. The GME 2000 metamodeling paradigm has a similar decorator as well. A decorator shell is also provided with this release to help you write your own decorators. Note that connection visualization has not changed and is not customizable.

Modeless dialog for attributes and preferences

The attributes and preferences dialogs have been merged into one tabbed dialog window that is always visible and dockable to the main window frame. These dialogs also display the object name. There are multiple ways to select the object whose attributes and preferences are shown. All the context menus (even from the browser now) provide access like before. If a new object is inserted, pasted or dropped, its attributes and preferences will be immediately shown. Finally, simply selecting an object by clicking on it, has the same effect. Note that currently the attributes and preferences dialog does not support multiple object selection.

Add-on and plug-in support

The add-on mechanism has been updated and tested. Whenever a data file is loaded, the activated add-ons are also loaded automatically. Add-ons listen to events; the event set listened to is specified through the component configurator GUI (ComponentConfig.exe) . For efficiency reasons, Add-ons cannot be Builder Object Network components. Plug-ins are now accessible through a separate command in the File menu.

OCL syntax checker for metamodeling

As a sample add-on, the metamodeling environment now comes with this nice helper tool. Every time a constraint expression attribute is changed this add-on is activated. Note that the target paradigm information is not available to this tool, therefore, it

cannot check arguments and parameters, such as kindname. These can only be checked at constraint evaluation time in your target environment.

Toolbar button/interpreter association capability

Interpreters and plug-ins can now register toolbar icons. An icon is either a resource in the component itself, or a separate icon file. If a project is loaded, the registered toolbar icons of all the active components are displayed in the toolbar, providing a user friendly way to start components. The icon information is stored in the registry under the 'Icon' field in the components registry node. Its format is either [<modulename>], <resource key>, or a full pathname of an image file.

Component interface version checking

Starting with version 1.2, GME 2000 is very conservative about component interface versions. First, the components that make up GME 2000 must always be present and have identical component interface version numbers. Components are also expected to be built against the very same interface as the GME 2000 executing them, although only a warning is displayed when starting incompatible components. There is no way to change the interface versions of compiled binaries. The only possible way to update the interface version number is to recompile the components against the up-to-date interface files.

Type inheritance refinements

The previous restriction, that only root models can be derived from or instantiated, has been relaxed. Now a model type can be derived or instantiated provided none of its ancestors or descendants (in the containment hierarchy) have any subtypes or instances.

Paste Special commands

Objects on the clipboard can now be pasted as references, subtypes or instances through these commands available through the regular menu and the context menus. References can also be redirected using the Redirection Paste command in the context menu. The usual restrictions still apply, i.e. paradigm violations and other illegal operations are not allowed. Note that the paste special commands only work if the source of the clipboard data is the same project open in the same GME 2000 instance.

Instant connections through context menus

In the regular edit mode connections can now be made by the Connect command in the context menu. Selecting this command changes the cursor to the connect cursor. A connection will be made to the object that is left clicked next. (Or by selecting the Connect command on the destination object as well.) Note that any other operation, such as mode change, window change, new object creation, cancels the connection operation.

Enhanced icon specification

GME icons settings now allow two macros, #PARADIGMDIR and $PROJECTDIR, that resolve to the directory of the current paradigm definition or project file, respectively.

Updated high-level C++ interpreter interface

(Builder Object Network or BON for short). BON now uses the IMgaComponentEx COM interface (also new in this version). The biggest change is that the Invoke function has been replaced by InvokeEx, which clearly separates the focus object from the selected objects. (Depending on the invocation method both of these parameters may be empty.) Components using the old BON will still work, however, upon invocation a warning is message displayed reminding users to upgrade the component code to fully comply with the new BON.

Tutorials

Three short, simple tutorials have been prepared on the metamodeling, metamodel composition and type inheritance.

A sample UML class diagram drawing paradigm

To illustrate the user-defined drawing capabilities of this version, we are including this simple paradigm as an example. Note that no interpreter is included.

Java high-level interpreter interface (alpha release)

We have prepared this interface that is very similar to the high-level C++ interface, the Builder Object Network (BON). This is an experimental version, not tested thoroughly. A fairly severe restriction is that it is based on Visual J++, because we have used the Java/COM bridge from Microsoft.

Introduction

The Generic Modeling Environment, GME 2000, is configurable model-integrated program synthesis tool.

The Generic Modeling Environment (GME 2000), is a Windows©-based, domain-specific, model-integrated program synthesis tool for creating and evolving domain-specific, multi-aspect models of large-scale engineering systems. The GME is *configurable,* which means it can be "programmed" to work with vastly different domains. Another important feature is that GMEs are *generated* from formal modeling environment specifications. This allows a particular GME to be efficiently designed and implemented, and ensures that it can be quickly and safely evolved as modeling requirements change.

The GME includes several other relevant features:

- It is used primarily for *model-building*. The models take the form of graphical, multi-aspect, attributed entity-relationship diagrams. The *semantics* of a model is not the concern of GME – that is determined later during the *model interpretation* process.
- It supports various techniques for building large-scale, complex models. The techniques include: hierarchy, multiple aspects, sets, references, and explicit constraints. These concepts are discussed later.
- It contains one or more integrated model *interpreters* that perform translation and analysis of models currently under development.

In this document we describe the commonalities of GME that are present in all manifestations of the system. Hence, we deal with general questions, and not domain-specific modeling issues. The following sections describe some general modeling concepts and the various functions of the GME.

Modeling Concepts Overview

Model-Integrated Program Synthesis

Model-integrated program synthesis is one method of performing model-integrated computing.

One approach to MIC is model-integrated program synthesis (MIPS). A MIPS environment operates according to a domain-specific set of requirements that describe how any system in the domain can be modeled. These modeling requirements specify the types of entities and relationships that can be modeled; how to model them; entity and/or relationship attributes; the number and types of aspects necessary to logically and efficiently partition the design space; how semantic information is to be represented in, and later extracted from, the models; analysis requirements; and, in the case of executable models, run-time requirements.

In MIPS, formalized models capture various aspects of a domain-specific system's desired structure and behavior. Model interpreters are used to perform the computational transformations necessary to synthesize executable code for use in the system’s execution environment–often in conjunction with code libraries and some form of middleware (e.g. CORBA, the MultiGraph kernel, POSIX) – or to supply input data streams for use by various GOTS, COTS, or custom software packages (e.g. spreadsheets, simulation engines) When changes in the overall system require new application programs, the models are updated to reflect these changes, the interpretation process is repeated, and the applications and data streams are automatically regenerated from the models.

Once a modeling paradigm has been established, the MIPS environment itself can be built. A MIPS environment consists of three main components: (1) a domain aware model builder used to create and modify models of domain-specific systems, (2) the models themselves, and (3) one or more model interpreters used to extract and translate semantic knowledge from the models.

The MultiGraph Architecture

MultiGraph *is a toolset for creating domain-specific modeling environments.*

The MultiGraph Architecture (MGA) is a toolset for creating MIPS environments. As mentioned earlier, MIPS environments provide a means for evolving domain-specific applications through the modification of models and re-synthesis of applications. We now discuss the creation of a MIPS environment.

The Modeling Paradigm

*A **modeling paradigm** defines the* family *of models that can be created using the resultant MIPS environment.*

The process begins by formulating the domain's *modeling paradigm*. The modeling paradigm contains all the syntactic, semantic, and presentation information regarding the domain – which concepts will be used to construct models, what relationships may exist among those concepts, how the concepts may be organized and viewed by the modeler, and rules governing the construction of models. The modeling paradigm defines the *family* of models that can be created using the resultant MIPS environment.

Both domain and MGA experts participate in the task of formulating the modeling paradigm. Experience has shown that the modeling paradigm changes rapidly during early stages of development, becoming stable only after a significant amount of testing and use. A contributing factor to this phenomenon is the fact that domain experts are often unable to initially specify exactly how the modeling environment should behave. Of course, as the system matures, the modeling paradigm becomes stable. However, because the system itself must evolve, the modeling paradigm must change to reflect this evolution. Changes to the paradigm result in new modeling environments, and new modeling environments require new or migrated models.

Metamodels and Modeling Environment Synthesis

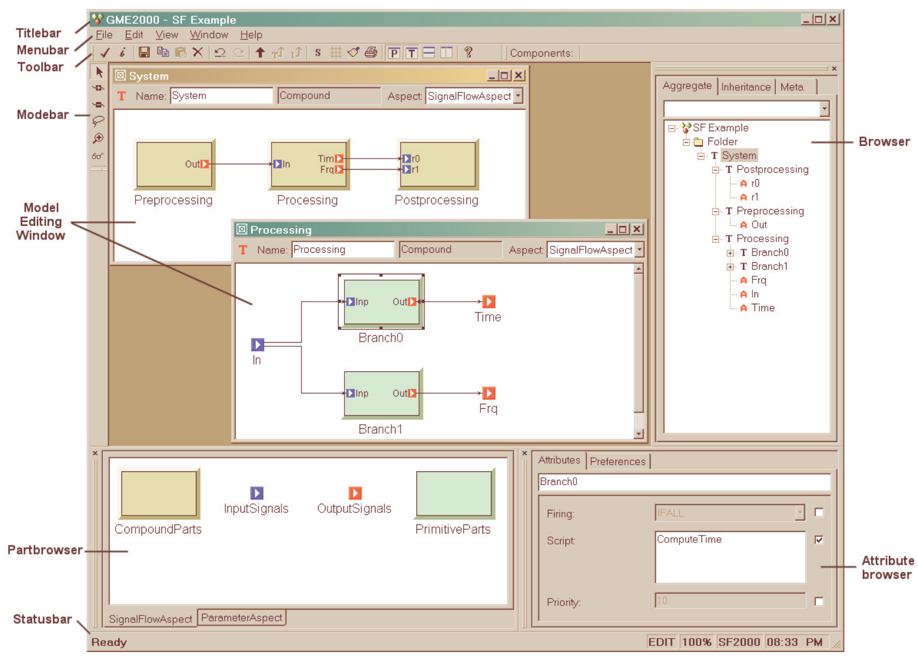
*A **metamodel** is a formalized description of a particular modeling language, and is used to synthesize the GME itself.*

Metamodels are models of a particular modeling environment. Metamodels contain descriptions of the entities, attributes, and relationships that are available in the target modeling environment. Once a metamodel is constructed, it is used to synthesize the actual GME. This approach allows the modeling environment itself to be evolved over time as domain modeling requirements change.

The Generic Modeling Environment

GME 2000 Main Editing Window

The figure below shows various features and components associated with the GME main editing window.



GME 2000 Main Editing Window

The GME main editing window has the following components:

- *Titlebar:* Indicates the currently loaded project.
- *Menubar:* Commands for certain operations on the model.
- *Toolbar:* Icon button shortcuts for several editing functions. Placing the mouse cursor over a toolbar button briefly displays the name/action of the button.

- *Mode bar:* Buttons for selecting editing modes.
- *Editing area:* The main model editing area containing the model editing windows.
- *Partbrowser:* Shows the parts that can be inserted in the current aspect of the current model.
- *Statusbar:* The line at the bottom which shows status and error messages, current edit mode (e.g. EDIT, CONNECT, etc.), zoom factor, paradigm name (e.g. SF2000), and current time.
- *Attribute browser:* Shows the attributes and preferences of an object.
- *Browser:* Shows either the aggregation hierarchy of the project, the type inheritance hierarchy of a model, or a quick overview of the current modeling paradigm.

These features will be described in detail in later sections.

GME Concepts

As mentioned above, the GME is a generic, programmable tool. However, all GME configurations are the same on a certain level, simply because "only" the domain-specific modeling concepts and model structures have changed. Before describing GME operation, we briefly describe the domain-independent modeling concepts embodied in all GME instances.

Defining the Modeling Paradigm

To properly model any large, complex engineering system, a modeler must be able to describe a system's entities, attributes, and relationships in a clear, concise manner. The modeling environment must constrain the modeler to create syntactically and semantically correct models, while affording the modeler the flexibility and freedom to describe a system in sufficient detail to allow meaningful analysis of the models. Issues such as what is to be modeled, how the modeling is to be done, and what types of analyses are to be performed on the constructed models must be formalized before any system is built. Such design choices are represented by the *modeling paradigm*. Therefore, creating the modeling paradigm is the first, and most important, step in creating a DSME.

A modeling paradigm is defined by the kind of models that can be built using it, how they are organized, what information is stored in them, etc. When GME is tailored for a particular application domain, the modeling paradigm is determined and the tool is configured accordingly. Typically the end-users do not change these paradigm definitions, and they are fixed for a particular instance of GME (of course, they may change as the design environment evolves).

Examples of modeling paradigms are as follows:

- Paradigms for modeling signal flow graphs and hardware architecture for high-performance signal processing domains.
- Paradigms for process models and equipment models used in chemical engineering domains.
- Paradigms for modeling the functionality and physical components of fault-modeling domains.

- Paradigms that describe other paradigms. These are referred to as *meta paradigms*, and are used to create *metamodels*. These metamodels are then used to automatically generate a modeling environment for the target domain.

Once an initial modeling paradigm has been formulated, an MGA expert constructs a metamodel. The metamodel is a UML-based, formal description of the modeling environment's model construction semantics. The metamodel defines what types of objects can be used during the modeling process, how those objects will appear on screen, what attributes will be associated with those objects, and how relationships between those objects will be represented. The metamodel also contains a description of any constraints that the modeling environment must enforce at model creation time. These constraints are expressed using the MultiGraph Constraint Language (MCL), a predicate logic language based on the Object Constraint Language (OCL) used with UML. Note that, as mentioned earlier, metamodels are merely models of modeling environments, and as such can be built using the GME. A special metamodeling paradigm has been developed that allows metamodels to be constructed using the GME.

Once a metamodel has been created, it is used to automatically generate a domain-specific GME. The GME is then made available to one or more domain experts who use it to build domain-specific models. Typically, the domain expert's initial modeling efforts will reveal flaws or inconsistencies in the modeling paradigm. As the modeling paradigm is refined and improved, the metamodel is updated to reflect these refinements, and new GMEs are generated.

Once the modeling paradigm is stable (i.e. the MGA and domain experts are satisfied that the GME allows consistent, valid models to be built), the task of interpreter writing begins. Interpreters are *model translators* designed to work with all models created using the domain-specific GME for which they were designed. The translated models are used as sources to analysis programs or are used by an execution environment.

Once the interpreters are created, environment users can create domain models and perform analysis on those models. Note, however, that model creation usually begins much sooner. Modelers typically begin creating models as soon as the initial GME is delivered. As their understanding of the modeling environment and their own systems grows, the models naturally become more complete and complex.

We now discuss the modeling components in greater detail.

Models

By *model* we mean an abstract object that represents something in the world. What a model represents depends on what domain we are working in. For instance,

- a Dataflow Block is the model for an operator in the signal processing domain,
- a Process model represents a functionality in a plant in the chemical engineering domain,
- a Network model represents a hardware interconnection scheme in the multiprocessor architecture domain.

A model is, in computational terms, an object that can be manipulated. It has *state*, *identity*, and *behavior*. The purpose of the GME is to *create and manipulate* these models. Other components of the MGA deal with *interpreting* these models and using them in various contexts (e.g. analysis, software synthesis, etc.).

Some modeling paradigms have several kinds of models. For instance:

- in a signal processing paradigm there can be Primitive Blocks for simple operators and Compound Blocks (which may contain both primitive blocks and other compound blocks) for compound operators.
- in a multiprocessor architecture modeling paradigm there can be models for computational Nodes and models for Networks formed from those nodes.

A model typically has *parts*—other objects contained within the model. Parts come in these varieties:

- atoms (or *atomic* parts),
- other models,
- references (which can be thought of as pointers to other objects),
- sets (which can contain other parts), and
- connections.

If a model contains parts, we say that the model is the *parent* of its parts. Parts can have various attributes. A special attribute associated with atomic parts allows them to be designated as *link* parts. Link parts act as connection points between models (usually used to indicate some form of association, relationship, or dataflow between two or more models). Models that can contain other models as parts are called *compound models*. Models that cannot contain other models are called *primitive models*. If a compound model can contain other models we have a case of model *hierarchy*.

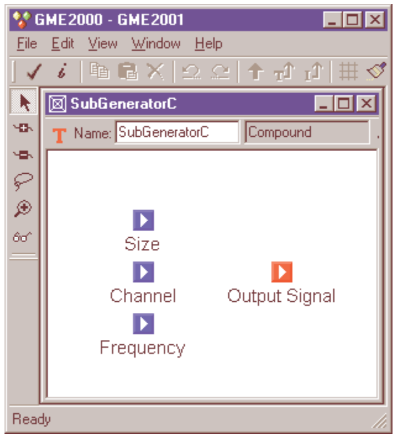
In the GME, each part (atom, model, reference, or set) is represented by an icon. Parts have a simple, paradigm-defined icon. If no icon is defined for a model, it is shown using an automatically generated rectangular icon with a 3D border.

Atoms

Atoms (or *atomic parts*) are simple modeling objects that do not have internal structure (i.e. they do not contain other objects), although they can have attributes. Atoms can be used to represent entities, which are indivisible, and exist in the context of their parent model.

Examples of atoms are as follows:

- An output data port on a dataflow block in a signal processing paradigm.
- A connection link on a processor model in a hardware description paradigm.
- A process variable in a process model in a chemical engineering paradigm.



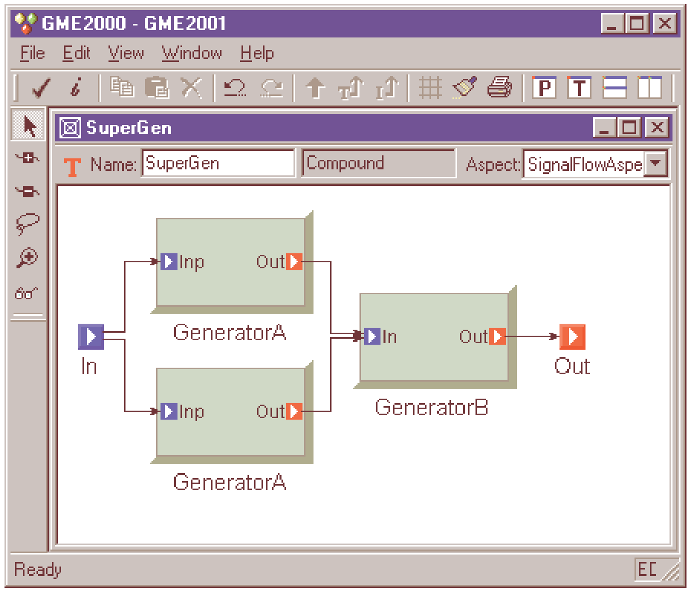
A primitive model `SubGeneratorC` containing four atoms

Model Hierarchy

As mentioned above, models can contain other models as parts—models of the same or different *kind* as the parent model. This is a case of model *hierarchy*. The concept can be explained as follows: models represent the world on different levels of abstraction. A model that contains other models represents something on a higher level of abstraction, since many details are not visible. A model that does not contain other models represents something on a lower level of abstraction. This hierarchical organization helps in managing complexity by allowing the modeler to present a larger part of the system, albeit with less detail, by using a higher level of abstraction. At a lower level of abstraction, more detail can be presented, but less of the system can be viewed at one time.

Examples where hierarchy is useful are as follows:

- Hierarchical dataflow diagrams in a signal processing paradigm.
- Process model hierarchy in a chemical engineering paradigm.
- Hierarchically organized networks of processors in a paradigm describing multiprocessors.



Compound model `SuperGen` containing several `Generator` models

References

References are parts that are similar in concept to pointers found in various programming languages. When complex models are created (containing many, different kinds of atomic and hierarchical parts), it is sometimes necessary for one model to directly access parts contained in another. For example, in one dataflow diagram a variable may be defined, and in another diagram of the system one may want to use that variable. In dataflow diagrams, this is possible only by connecting that variable via a dataflow arc, "going up" in the hierarchy until a level is reached from where one can descend and reach the other diagram (a rather cumbersome process).

GME offers a better solution – *reference parts*. Reference parts are objects that refer to (i.e. *point* to) other modeling objects. Thus, a reference part can point to a model, an atomic part of a model, a model embedded in another model, or even another reference part or a set. A reference part can be created only after the referenced part has been created, and the referenced part cannot be removed until all references to it have been removed. However, it is possible to create null references, i.e. references that do not refer to any objects. One can think of these as placeholders for future use. Whether a particular reference can be established (i.e. created) or not depends on the particular modeling paradigm being used.

Examples of references are as follows:

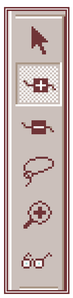
- References to variables in remote dataflow diagrams in a signal processing paradigm.
- References to equipment models in a process model in a chemical engineering paradigm.
- References to nodes of a multiprocessor network in a paradigm describing hardware/software allocation assignments.

As mentioned above, the icon used to represent the reference part is user-defined. Model (or model reference) references that do not have their own icon defined have an appearance similar to the referred-to model, but without 3D borders.

Connections and links

Merely having parts in a model is not sufficient for creating meaningful models—there are relationships among those parts that need to be expressed. The GME uses many different methods for expressing these relationships, the simplest one being the *connection*. A connection is a line that connects two parts of a model. Connections have at least two attributes: *appearance* (to aid the modeler in making distinctions between different types of connections) and *directionality* (as distinguished by the presence or absence of an arrow head at the "destination" end of the line). Additional connection attributes can be defined in the metamodel, depending on the requirements of the particular modeling paradigm.

The actual semantics of a connection is determined by the modeling paradigm. When the connection is being drawn, the GME checks whether the connection is legal or not. All legal connections are defined in the metamodel. Two checks are made to determine the legality of a connection. First, a check is made to determine if the two types of objects are allowed to be connected together. Second, the *direction* of the connection needs to be checked.



GME edit mode bar with the "Connections" mode button selected.

To make connections, the modeler must place the GME in the "Add Connections" mode. This is done by clicking on the "Connections" mode button (see figure to left) on the edit mode bar. A connection always connects two parts. If the part is an icon that represents a model, it may have some connection points, or *links*. Logically, a link is a port through which the model is connected to another part *within the parent model*. Links on a model icon represent specific parts contained in the model that are involved in a connection. In these cases, when the connection is established, care should be taken to build the connection with the right link. The link shows up on the icon of the model part as a miniature icon with a label. When the connection is built, the system uses these miniature icons as sensitive "pads" where connections may start or end. Moving the mouse cursor over one of the pads shows the complete name of the link part. Unlike with earlier version of GME 2000, connections can be made directly to models, not only its ports (if the modeling paradigm defined by the metamodel allows it). Furthermore, not only atoms, but models, sets and references except for connections can act as a ports. This is again a new feature in version 1.1.

Some examples of connections and links are as follows:

- Connections between dataflow blocks in a signal processing paradigm.
- Connections between processes on a process flow sheet of a chemical engineering paradigm.
- Connections between failure modes (indicating failure propagation) in a fault modeling paradigm.

Connections can be seen between atomic parts and models, as in the case of the `Input Signal` atomic part connecting to the ports labeled "Inp" on each of the `Generator` models shown earlier, and between ports of models, as in the case of the "`Out`" ports of each `Generator` model connecting to the "`Inp`" port of another `Generator` model. Notice that, in this paradigm, are directional (used to indicate information flow between the models).

Attributes

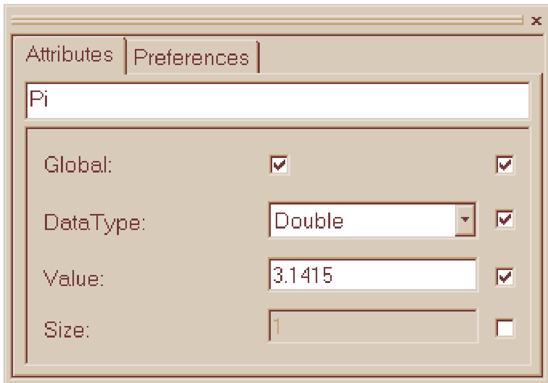
Models, atoms, references, sets and connections can all have *attributes*. An attribute is a property of an object that is best expressed textually. (Note that we use the word

"text" for anything that is shown as text, including numbers, and a choice from a finite set of symbolic or numeric constants.)

Typically objects have multiple attributes, which can be set using "non-graphical" means, such as entry fields, menus, buttons, etc. The attribute values are translated into object values (e.g. numbers, strings, etc.) and assigned to the objects. The modeling paradigm defines what attributes are present for what objects, the ranges of the attribute values, etc. GME does not interpret these values—this task is left to the model interpreters.

Examples of attributes are as follows:

- Data type of parameters in a signal processing paradigm.
- Units for process parameters in a chemical engineering paradigm.
- Mean-time-between-failure specifications for components in a fault modeling paradigm.



The attribute box associated with a `Parameter` atom called Pi.

An object's attributes can be accessed by right-clicking on the object and selecting "Attributes" from the menu, causing the object's attributes box to pop up. (The use of checkboxes that appear to the right of attribute controls is related to type inheritance and will be discussed later.)

Aspects

As mentioned earlier, we use hierarchy to show or hide design detail within our models. However, large models and/or complex modeling paradigms can lead to situations where, even within a given level of design hierarchy, there may be too many parts displayed at once. To alleviate this problem, models can be partitioned into *aspects*.

An aspect is defined by the kinds of parts that are visible in that aspect. Note that aspects are related to *groups* of parts. The existence or visibility of a part within a particular aspect is determined by the modeling paradigm. A given part may also be visible in more than one aspect. For every kind of part, there are two kinds of aspects: primary and secondary. Parts can only be added or deleted from the model from within its primary aspect. Secondary aspects merely *inherit* parts from the primary aspects. Of course, different interconnection rules may apply to parts in different aspects.

When a model is viewed, it is always viewed from one particular aspect at a time. Since some parts may be visible in more than one aspect while others may visible only in a single aspect, models may have a completely different appearance when viewed from different aspects (after all, that's why aspects exist!)

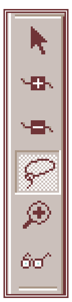
The following are examples of aspects:

- "Signal Flow" and "States" aspects for a signal processing paradigm.
- "Process Flow Sheet" and "Process Finite State Machine" aspects for a chemical engineering paradigm.
- "Component Assignment" and "Failure-Propagation" aspects of a fault-modeling paradigm.

Sets

Models containing objects and connections show a static system. In some cases, however, it is necessary to have a model of a *dynamic* system that has an architecture that changes over time. From the visual standpoint this means that, depending on what "state" the system is in, we should see different pictures. These "states" are not predefined by the modeling paradigm (in that case they would be aspects), but rather by the modeler. The different pictures should show the same model, containing the same kinds of parts, but some of the parts should be "present" while others should be "missing" in a certain "states." In other words, the modeler should be able to construct sets and subsets of particular objects (even connections).

In the GME, each set is represented by an icon (user-defined or default). When a particular set is activated, only the objects belonging to that set are visible (all other parts in the model are "dimmed" or "grayed out.") Parts may belong to a single set, to more than one set, or to no set at all.



GME edit mode bar with the "Set" mode button selected.

To add or remove parts from sets, the set must first be activated by placing the graphical editor into Set Mode. This is done by clicking the "Set Mode" button (see left) on the edit mode bar. Next, a set is activated by right-clicking the mouse on it. Once the set has been activated, parts (even connections) may be added and/or removed using the left mouse button. To return to the Edit Mode, click the "Normal Mode" button on the edit mode bar.

The following examples of using sets:

- State-dependent configuration of processing blocks in a signal processing paradigm.
- State dependent process configuration in a chemical engineering paradigm.
- State-dependent failure propagation graphs in a fault modeling paradigm.

Preferences

Preferences are paradigm-independent properties of objects. The five different kinds of first class objects (model, atom, reference, connection,

set) each have a different set of preferences. The most important preference is the help URL. Others include color, text color, line type, etc. Preferences are inherited from the paradigm definition through type inheritance unless this chain is explicitly broken by overriding an inherited value. For more details, see the chapter on type inheritance.

Preferences are accessible through the context menus and for the current model through the Edit menu.

Default preferences can be specified in the paradigm definition file (XML). User settings can be applied to either the current object, or the *kind* of object globally in the project. The last item in the preferences dialog box specifies this scope information. If the global scope is selected, the information is stored in the compiled, binary paradigm definition file, not in the XML document. This means that a subsequent parsing of the XML file overwrites preference settings. This limitation will be eliminated in a later release of GME 2000.

Even when the global scope is selected, this only applies to objects that themselves (or any of their ancestors) have not overridden the given preference.

Using GME 2000

GME 2000 Interfaces

The GME interacts with the user through two major interfaces:

- the **Model** Browser, and
- the **Graphical Editor**.

Models are stored in a model database and are organized into *projects*. A project is a group of models created using a particular modeling paradigm. Within a project, the models are further organized into modeling *folders*. Folders themselves and models in one folder can be organized hierarchically, although standalone models can also be present.

The Model Browser is used to view or look at the entire project “at a glance.” All models and folders can be shown, and folders, models and any kind of parts can be added, moved, and deleted using the Model Browser controls. This is described in more detail below.

The Part Browser

The Part Browser window shows the parts that can be inserted into the current model in the current aspect. It shows all parts except for connections. At the bottom of the Part Browser, tabs show the available aspects of the current model. Clicking on a tab will change the aspect of the current model to the selected one. It also attempts to change the aspect of all the open models. If a particular model does not have the given aspect, its current aspect remains active.

The Part Browser can be used to drag a single object at a time and drop it either in any editor window or in the Model Browser. If a reference is dragged, a null reference is created because the target object is unspecified. Remember that references (null references included) can be redirected at any time by dropping a new target on top of them (see more detailed discussion where the drag and drop operations are described).

Note that the Part Browser window, just like the Model Browser window, is dockable; it can float as an independent window or it can be docked to any side of the GME 2000 main window.

The Attribute Browser

Attributes and preferences are now available in a modeless dialog box, called the Attribute Browser. Since there is no OK button, changes are updated immediately. More precisely, changes to toggle buttons, combo boxes (i.e. menus) and color pickers are immediate. Changes to single line edit boxes are updated when either "Enter" is hit on the keyboard or the edit box loses the input focus, i.e. you click outside the box. The only difference for multiline edit boxes is that they use the Enter key for new line insertion, so hitting it does not updated the value.

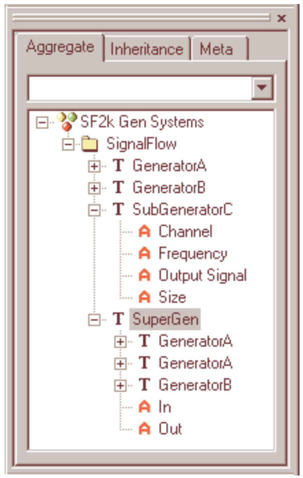
The object selection for the attribute browser works as follows. The context menu access to attributes and preferences, now even from the Browser, works. Furthermore, simply selecting an object or inserting, dropping or pasting it selects that object for the Attribute browser.

At the top of the dialog there are two tabs, one for the attributes and one for the preferences. Note that the Attribute Browser window, just like the Model Browser window, is dockable; it can float as an independent window or it can be docked to any side of the GME 2000 main window.

The Model Browser

As mentioned earlier, the GME is a configurable graphical editing environment. It is configured to work within a particular modeling paradigm via a *paradigm definition file*. Paradigm definition files are XML files that use a particular, GME 2000 specific Document Type Definition (DTD). Models cannot be created and edited until a paradigm definition file (or its compiled, binary version with *.mta* extension) has been opened.

Once a project has been loaded, the GME opens a *Model Browser* window. The Model Browser is primarily used to organize the individual models that make up an overall project, while the graphical editor is used for actually constructing the project's individual models.



Model Browser showing folders and models.

The most important high-level features of the Browser are accessible through the three tabs displayed at the top of the Browser. These tabs deal with the Aggregate, Inheritance, and Meta hierarchies.

The Aggregate tab contains a tree-based containment hierarchy of all folders, models, and parts from the highest level of the project, the Root Folder. The aggregate hierarchy is ignorant to aspects, and is capable of displaying objects of any kind. More information on the aggregate hierarchy will be provided shortly.

Model Browser with each tab selected

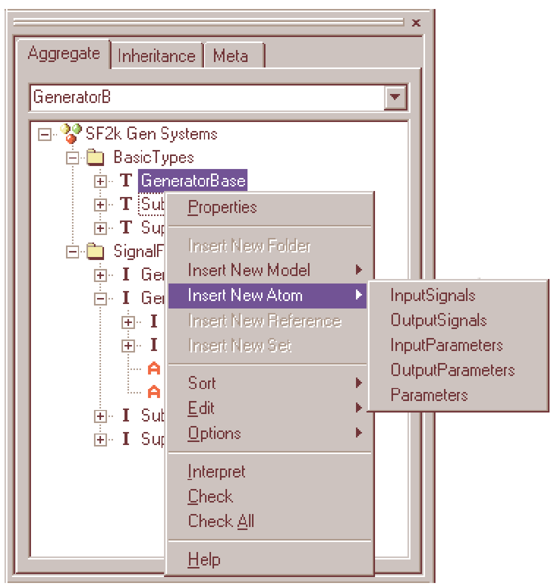
The Inheritance tab is used explicitly for visualizing the type inheritance hierarchy (described in detail later). It is entirely driven by the current model selection within the aggregate tree. For example, the current selection in the aggregate tree in the figure above is a model "GeneratorB". It is actually an instance of the model shown in the "BasicTypes" folder in the aggregate tree bearing the name "GeneratorBase". This type/instance relationship is shown in the Inheritance tab. Tracing up the hierarchy we have the "GeneratorBase" model. Beneath it we display a folder containing any instances of this type, and then any derived versions of this type. The "GeneratorB" model highlighted in the Aggregate tree that we are referring to is contained in this folder. Alongside that is another instance of the same Generator type. On the same level as the Instances folder for the GeneratorBase, we have a subclass of that, and relevant Instances.

The Meta tab shows the modeling language at a glance: it displays the legally available array of Folders and objects that can be added at any level within the aggregate hierarchy. For example, at the "Root Folder" level we can add "SignalFlow" folders. Within "SignalFlow" folders, we can add models Primitive and Compound. From these models, more parts can be added.

Model Browser navigation

Arrow keys can navigate the selection in vertical directions. The Delete and Backspace keys allow for deletion of the current selection. Object name editing is achieved through delayed clicking on an object's name. Multiple selection is achieved through <shift> or <control> clicks. Note: parents of parts cannot be selected simultaneously with their children. Smart selection will either avoid the parent or avoid the parts based on the first selected item in this selection context. Incremental searching is offered for all three tabs through the text entry field immediately below the Aggregate, Inheritance, and Meta tab selections. The search is limited to the currently expanded section of the tree to avoid time-consuming search in a potentially large database. If a global search is desired, pressing the Asterisk key when the root folder is selected fully expands the tree and the search becomes project-wide.

Most hidden functionality offered within the GME 2000 Browser is available through contextual menus and drag and drop operations. Currently contextual menus are only offered for selections found within the Aggregate tab. Contextual information is primarily used for easily inserting new objects based on the current selection, or for capturing the contents of current selections for Edit functions (Copy, Paste, Delete, etc.).



Model Browser context menus

Based on the Aggregate tab selection shown above, five different kinds of atoms are available for insertion (Models can also be inserted, but within this Model we have specified that the paradigm not allow any References or Sets). Note that connections cannot be added using the Browser.

Similarly, several Edit options are available in the form of Undo, Redo, Copy, Paste, etc. Sorting options allow for the all of the objects and their children to be sorted by a specific style. The Options/Display submenu displays a dialog used for specifying

the types of objects to be displayed in the Aggregate Tab. For example, the user can choose not to view connections in the browser. Interpreting, Constraint Checking, and context sensitive Help are also available.

Drag and drop is implemented in the standard Windows manner. Multiple selection items may serve as the source for Drag and Drop. Modifiers are important to note for these operations:

- No modifier: Move operation
- Ctrl: Copy (signified by "plus" icon over mouse cursor)
- Ctrl+Shift: Create reference (signified by link icon over mouse cursor)
- Alt: Create Instance (signified by link icon over mouse cursor)
- Alt+Shift: Create Sub Type (signified by link icon over mouse cursor)

If a drop operation fails, then a dialog will indicate so. Drop operations can occur within the Browser itself, allowing this to be an effective means to restructuring a hierarchy. Drop operations can only be performed onto a Model or a Folder.

Model Browser and Interoperation

Double-clicking on any model in the tree (or pressing the Space Bar or Enter key when a model is selected) will open that model for editing in the graphical model editor. Double-clicking an atom, reference or set, will open up the parent model, select the given object and scroll the model, so that the object becomes visible.

Locking

Using the MS Repository backend, distributed multi-user access is allowed to the same project. To ensure consistency, GME 2000 implements a sophisticated locking mechanism.

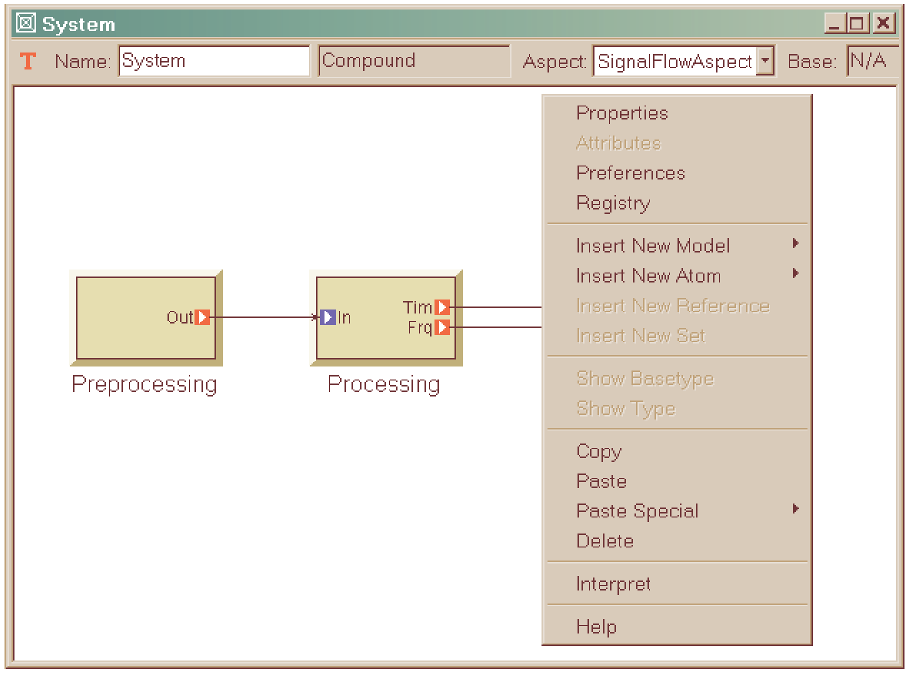
There are four different types of locks from the perspective of a user. An object can be *not locked*, *read-only* locked, *write-only* locked or *exclusively* locked. When an object is read-only locked, then other users may access the same object, but only in read-only mode. The read-only lock guarantees that all information read from the object is up-to-date and cannot be modified by other users while the lock is held. When an object is write-only locked, then others can still access the same object write-only, but not read-only or exclusively. The write-only lock guarantees that the object is kept modifiable, while the write-only lock is held. It gives no guarantee, however, that any information read from the object is up-to-date. Reference objects are the prime reason for introducing the write-only lock. Multiple users must be allowed to make references to the same target model. To make matters worse, different users have different undo queues, possibly containing modifications to the same objects. Holding a write-only lock on the target model and exclusive locks on the referee objects solves this problem. Finally, an exclusive lock is equivalent to holding read-only and write-only locks simultaneously.

In summary, an object is either not locked at all, read-only locked by a few users, write-only locked by a few users, or exclusively locked by a single user. Note that the object lock states are visualized in the Model Browser with an additional icon displayed next to the regular one.

The Model Editor

The Editing Window

When a model is selected for editing, an *editor* window opens up to allow editing of that model. The editor window shows the contents of the selected model in one aspect at a time.



A typical model editing window with an open context menu.

A typical model editing window is shown above. The status line near the top begins with an icon indicating whether the current model is a type (T) or instance (I). Next to it is a field indicating the model's name – `System` in this case. Next to the model's name is the *kind* field, indicating the kind of model (e.g. `Connector`, `Compound`, `Network`, etc.) being edited. Continuing to the right, the `Aspect` field indicates that this model is being viewed in the `SignalFlowAspect`. Remember, a model's appearance, included parts, and connection types can change as different aspects are selected. Finally, the right side of the status line shows the base type of this model in case it is a model type (if it is an archetype, it does not have a base type, so the field shows N/A), or the type model in case the current model is an instance.

GME Menus

On the GME Menubar, the following commands are available:

File: Project- and model-related commands.

The File menu is context-sensitive, with choices depending on whether or not a paradigm definition file and/or project has been loaded and whether there is at least one model window open. If no model window is open, the following items show:

- **New Project:** Creates a new, empty project and allows registering a new modeling paradigm (discussed in detail later).
- **Open Project:** Opens an existing project from either a database or a binary file with the *.mga* extension (discussed in detail later).
- **Close Project:** Saves and closes the currently open project (if any).
- **Save Project:** Saves the current project to disk.
- **Save Project As:** Saves the current project with a new name.
- **Abort Project:** Aborts all the changes made since last save and closes project.
- **Export XML:** GME 2000 uses XML (with a specific DTD) as a export/import file format. This command saves the current project in XML format.
- **Import XML:** Loads a previously exported XML project file. Note that the file must conform to the DTD specifications in the mga.dtd file. If no paradigm is loaded, GME 2000 tries to locate and load the corresponding paradigm definitions.
- **Update through XML:** Allows updating the current model in case of a paradigm change. If the user has a project open in one GME 2000, while she modifies the metamodels in another GME 2000 and regenerates the paradigm, this command allows updating the models by automatically exporting toXML and importing from it. Note that any changes that invalidate the existing models, for example deleting a model kind that has instances in the project, will cause this operation to fail. However, adding new kinds of objects, attributes, etc, or deleting unused concepts will work.
- **Register Paradigms:** Registers a new modeling paradigm (discussed in detail later).
- **Register Component:** Registers an interpreter DLL with the current paradigm. A dialog box appears that makes it possible to register as many interpreters as the user wishes.
- **Check All:** Invokes the Constraint Manager to check all constraints for the entire project.
- **Settings:** Sets GME 2000-specific parameters. Currently, the only supported option is to set the path where the icon files are located on the current machine. The user can type in a semicolon separated list of directories (the order is significant from left to right), or use the add button in the dialog box to add directories one-by-one utilizing a standard Windows File Dialog Box. Icon directories can be set for system-wide use or for the current user only. GME 2000 searches first in the user directories followed by the system directories.
- **Exit:** Closes GME 2000.

Once a model window is open, the following additional items become available:

- **Run Interpreter:** As mentioned earlier, model interpreters are used in the GME to extract semantic information from the models. This menu choice invokes the model interpreter registered with the paradigm using the currently selected model as an argument. Depending on the specific

paradigm and interpreter, such an argument may or may not be necessary. A submenu makes it possible to select an interpreter if there is more than one interpreter available.

- **Run Plug-Ins:** Plug-ins are paradigm independent interpreters. This command makes it possible to run the desired one.
- **Check:** Invokes the Constraint Manager to check the constraints for the current model.
- **Print:** Allows the user to print the contents of the currently active window. It scales the contents to fit on one page.
- **Print Setup…:** Standard Windows functionality.

After a project has been loaded or created, the following menu items are active:

Edit: Editing commands.

- **Undo, Redo**: The last ten operations can be undone and redone. These operations are project-based, not model/window-based! The Browser, Editor, and interpreters share the same undo/redo queue.
- **Clear Undo Queue**: Models that can be potentially involved in an undo/redo operation are locked in the database (in case of a database backend, as opposed to the binary file format), so that no other user can have write access to them. This command empties the undo queue and clears the locks on object that are otherwise not open in the current GME 2000 instance.
- **Project Properties**: This command displays a dialog box that makes it possible to edit/view the properties of the current project. These properties include its name, author, creation and last modification date and time, and notes. The creation and modification time stamps are read-only and are automatically set by GME 2000.

Items available only when a model window is open:

- **Show Parent**: Active when the current model is contained inside another model. Selecting this option opens the parent model in a new editing window.
- **Show Basetype**: Active when the current model is a type model but not an archetype (i.e. it is not a root node in the type inheritance hierarchy). This command opens the base type model of the current model in an editing window.
- **Show Type**: Active when the current model is an instance model. This command opens the type model of the current model in an editing window.
- **Copy, Paste, Delete, Select All**: Standard Windows operations.
- **Paste Special**: A submenu makes it possible to paste the current clipboard data as a reference, subtype or instance. Paste Special only works if the data source is the current project and the current GME 2000 instance.
- **Cancel**: Used to cancel a pending connect/disconnect operation.
- **Preferences**: Shows the preferences available for the current model (see detailed discussion in a separate section below).

- **Registry**: The registry is a property extension mechanism: any object can contain an arbitrarily deep tree structure of simple key-value pairs of data. Selecting this menu item opens up a simple dialog box where the current object's registry can be edited. Special care must be taken when editing the registry, since it is being used by the GME 2000 GUI to store visualization information and domain-specific interpreters may use it too.
- **Synch Aspects**: The layout of objects in an aspect is independent of other aspects. However, using this functionality, the layout in one source aspect can be propagated to multiple destination aspects. A dialog box enables the selection of the source and destination aspects. The objects that participate in this operation can also be controlled here. The default selection is all the visible objects in the source aspect if none of them were selected in the editing window, otherwise, only the selected ones. Two check boxes control the order in which objects are moved. This is important in case objects compete for the same real estate. Priority can be given to the selected objects and within the selected objects the ones that are visible in the source aspect.

View: Allows the toggling on and off of the Toolbar, the Status Bar (bottom of the main window), the Browser window, the Attribute Browser, and the Part Browser window.

Window:

- **Cascade, Tile, Arrange Icons:** Standard Windows window management functions.

Help:

- **Contents**: Accesses the ISIS web server and shows the contents page of this document.
- **Help**: Shows context-sensitive, user-defined help (if available) or defaults to the appropriate page of this document. See details in a subsequent section.
- **About**: Standard Windows functionality.

Managing Paradigms

The Register Paradigm item in the File menu displays a dialog box where the user can add or modify paradigms. This dialog box is also displayed as the first step of the New Project command (see below).

Like other items recorded in the Windows registry, paradigms can be registered either in the current user's own registry [HKEY_CURRENT_USER/Software/Mga 2000/Paradigms] or in the common system registry [HKEY_CURRENT_SYSTEM/Software/Mga 2000/Paradigms]. If a paradigm is registered in both registries, the per-user registry takes precedence. When changing the registration of paradigms it can be specified where the changes are to be recorded. Non-administrator users on Windows 2000 systems generally do not have write access to the system registry, so they can only change the per-user registration.

Paradigms are listed by their name, status, connection string and current version ID. The name is what primarily identifies the paradigm. The status is 'u' (user) or 's' (system) depending where the paradigm is registered. The connection string specifies

the database access information or the file name in case of binary files. Version ID is the ID of the current generation of the paradigm.

The registry access mode is selectable in the lower right corner of the dialog box.

Pressing the "Add from file…" button displays a file dialog where the user can select compiled binary files (.mta) or XML documents. It is possible to store paradigm information in MS Repository as well. The "Add from DB…" is used to specify paradigms stored in a database, like MS Access.

If the new paradigm specified was not yet registered, it will be added the list of paradigms. If, however, the paradigm is an update to an existing paradigm, it will replace the existing one, but the old paradigm is also kept as a previous generation. (The only exception is when the paradigms are specified in their binary format (i.e. not XML) and the file or connection name of the new generation corresponds to that of the previous one.) This way existing models can still be opened with the legacy paradigms they were created with. For new models, however, the current generation is used always.

Paradigms can be unregistered using the Remove button. Note that the paradigm file is not deleted.

Different generations of an existing paradigm can be managed using the Purge/Select button. This brings up another dialog showing all the generations of the selected paradigm. One option is to set the current generation, the one used for creating new models. The other option allows unregistering or also physically deleting one or several of the previous generations. (Whether the files are deleted is controlled by the checkbox in the lower right corner.)

IMPORTANT! New paradigm versions are not always compatible with existing binary models. If a model is reopened, GME offers the option to upgrade it to the new paradigm. If the upgrade fails, XML export and re-import is needed (the previous generation of the paradigm is to be used for export). XML is usually the more robust technique for model migration; it only fails if the changes in the paradigm make the model invalid. In such a situation the paradigm should be temporarily reverted to support the existing model, edited to eliminate the inconsistencies, and then reopened with the final version of the paradigm.

New Project

Selecting the New Project item in the file menu displays the dialog box described in the previous section. All the features mentioned are available, plus an additional button, 'Create New...', which is used to proceed with the creation of a new project.

Once the desired paradigm is selected, pressing the OK button displays another small dialog where the user can specify whether to store the new project in MS Repository or a binary file. Pressing OK creates and opens a new blank project. At this point, the only object available in the project is the root folder shown in the Model Browser. Using the context menu (right-clicking the Project Name), the user can add folders and other objects, as defined in the paradigm. Double-clicking a model opens it up in a new editor window.

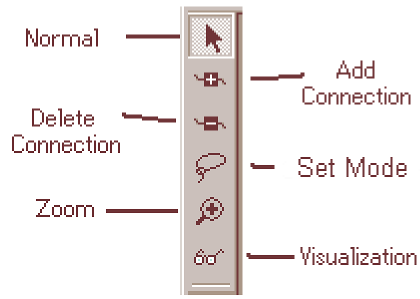
Editor Operations

Using the editor window the user can edit the models graphically. Menus and editing operations are context sensitive, preventing illegal model construction operations. (Note, however, that even a syntactically correct model can be invalid semantically!)

This section gives a brief overview of common editor operations, such as changing editing modes, creating and destroying models, placing parts, etc.

Editing Modes

The graphical editor has six editing modes – Normal, Add Connection, Delete Connection, Set Mode, Zoom Mode and Visualization. The Editing Mode Bar, located (by default) just to the left of the main editing window, is used to change between these modes.



GME Editing Mode Bar

The figure above indicates the buttons used to select different editing modes. The Editing Mode Bar is a *dockable* Windows menu button bar. It can be dragged to different positions in the editor, floated on top of the editing window, or docked to the side of the editor.

Normal Mode

Normal mode is used to add/delete/move/copy parts within editing windows. Models (from the Model Browser) and parts (from the Part Browser) may be copied by left-click-dragging the objects into the editing window. Standard Windows keyboard shortcuts (Ctrl-C to Copy, Ctrl-V to Paste) may also be used. A copy operation (the default when dragging from the Part Browser) is indicated by the small "+" symbol attached to the mouse cursor during the left-click-drag operation.

Parts and models may be moved and/or copied between models, too. Here, the normal left-click-dragging operation causes a *move* operation instead of a copy. To copy parts and models between or within models, hold down the Ctrl key before dropping.

New parts and models are given a default name (defined in the modeling paradigm). Right-clicking a part (even connection) brings up a context menu. Choose Properties to edit/view an object's properties. Choose Attributes to edit its paradigm-specific attribute values.

Right-clicking on the background of a model window brings up another context menu that makes it possible to insert any part that is legal in the current aspect of the given model.

As mentioned earlier, reference parts act as pointers to objects, providing a *reference* to that part or model. References are created by holding down Ctrl-Shift while dropping parts into a new model from another model window or from the Browser. When dragging a reference from the part browser it is not necessary to hold down any keys because the source already specifies that a reference is to be created. In this case, however, a null reference is created since there is no target object specified (similar to using the context menu to insert a reference).

References can be redirected, i.e. the object they refer to can be changed. Simply drop an object on top of an existing reference, and if the object kind matches, the reference is redirected. Note that the type hierarchy places restrictions on this operation as well (see later in the Type Inheritance chapter).

Subtypes and instances of models can be created by holding down Alt+Shift and Alt keys respectively during the drop operation. Type inheritance is described in a separate chapter.

Parts and models may be removed by left-clicking to highlight them, and either selecting Delete from the Edit menu, or by pressing the Del keyboard key. Note that any connections attached to an object will also be deleted when that part or model is deleted. Also remember that parts can only be deleted after all references to them have already been deleted.

Add Connection Mode

This mode allows connections to be made between modeling objects. Connections may exist between two atomic parts, between two model ports (think of these as connection points on models), or between an atomic part and a model port. Remember, however, that connections are a paradigm-specific notion and will only be allowed between objects specified by the paradigm definition file as being allowed to be connected together.

Remember that connections are inherently directional in nature. Connections are made by first placing the editor in the Add Connection Mode, then left-clicking the source object, followed by left-clicking on the destination object.

It is not necessary to go to this mode to create a connection. Instead, in Edit mode right clicking on the desired source of a new connection and selecting Connect in the context menu changes the cursor to the connect cursor. A connection will be made to the object that is left clicked next. (Or by selecting the Connect command on the destination object as well.) Note that any other operation, such as mode change, window change, new object creation, cancels the instant connection operation.

Remove Connection Mode

By placing the graphical editor in the Remove Connection Mode, connections between objects can be removed by simply left-clicking on the connection itself or the source and/or destination parts.

Set Mode

Set parts are added to a model just like any other part. However, their members can only be specified when the editor is in Set Mode. Once the editor is in this mode, right-clicking a set will cause all parts (even connections) in the model that are not part of the given set to be "grayed out." Left-clicking object toggles their membership in the set. As they are added/removed to the set, they regain/lose their color and appearance.

Zoom Mode

The Zoom Mode allows the user the view the models at different levels of magnification. The supported range is between 10% and 300%. Left clicking anywhere in a model window zooms in, while right-clicking zooms out. The zoom level is window-specific.

Visualization Mode

The Visualization Mode allows single objects and collections of objects ("neighborhoods" of objects) to be visually highlighted with respect to other modeling objects. This is useful when examining and/or discussing complex models.

To enter the Visualization Mode, select the Visualization Mode button on the GME editing mode bar (see picture above). This will cause all visible parts and connections to become "grayed out." Next, the user may click on objects using either the left or right mouse buttons to make them fully visible again. Left- and right-clicking have different effects, as described below.

Left-clicking on any part toggles the visibility of the object. For connections, their source and destination objects are toggled. The user may continue to select parts in this manner, highlighting/hiding more and more objects. Right-clicking on a part will toggle the visibility of the object and the objects at the ends of its connections. Note that exactly those connections are highlighted at any one time that connect highlighted objects.

Miscellaneous operations

The following operations are only accessible from the toolbar:

- Toggle grid: At zoom levels 100% or higher a grid can be displayed in the model editor window. GME objects always snap to this fine grid, whether they are visible or not, to facilitate alignment of the objects.
- Refresh: Clicking the paintbrush button forces GME 2000 to repaint all the windows.

In the current model editing window there is a selected list of objects highlighted by little frames. Using the Arrow keys on the keyboard, these objects can be nudged by one grid cell in the selected direction, provided that there are no collisions. Note that GME 2000 does not allow overlapping objects.

Connections in GME 2000 are automatically routed. The user only needs to specify the end points of a connection and an appropriate route will be automatically generated that will avoid all objects and try to provide a visually pleasing connection layout.

The built-in context-sensitive help functionality is described in the next section.

Help System

GME 2000 provides context-sensitive, user-defined help functionality. This is facilitated by the "Help URL" preference of objects. This preference is inherited from the paradigm definition and through the type inheritance hierarchy exactly like any other object preference. For more information on this inheritance, see the separate chapter on type inheritance.

When the user selects help on a context menu or the Help menu Help item for the current model (also the F1 key), GME 2000 looks up the most specific help URL available for the given object. If no help URL is found, the program defaults to the appropriate section of the User's Manual located on the ISIS web server.

When the appropriate URL is located, GME 2000 invokes the default web browser on the current machine and displays the contents of the URL. If no network

connection is available, the help system will be unable to display the information unless the web server is running on the current machine or the URL refers to a local file.

Constraint Manager

GME 2000 includes an integrated Constraint Manager (CM) component. The task of the CM is to enforce the constraints that the given paradigm contains. All the constraints are checked when the user initiates constraint checking through the File menu or the toolbar. The Check item and the checkmark looking toolbar button checks all the constraint that correspond to the current model (i.e. the one being shown in the currently active window). The Check All item makes the CM check all the constraints for all the objects in the project.

Constraints can also be associated with one or more events, such as Attribute Change, Close Model etc. When this events occur, the CM checks all the constraints that are registered for the given event for the object that caused the event.

Each constraint has a priority. The priority controls the order of constraint evaluation. Higher priority constraints are checked first. The priority also controls what action is taken if a constraint is violated. If the priority is the highest (1) then an error message is displayed and the current transaction is aborted. In other words the operation that caused the violation is aborted, the models remain consistent with the constraint. If the priority is smaller (i.e. greater than 1), the user gets a warning message and she can choose to continue evaluating constraints, skip the remaining constraint or abort the transaction.

Constraints are specified as part of the meta representation. The constraint specification contains the constraint equation written in MCL, a language based on the Object Constraint Language (OCL), the priority, a short description of the constraint that is displayed as part of the error/warning message in case of a violation. For a detailed description of MCL, see Appendix B.

Type Inheritance

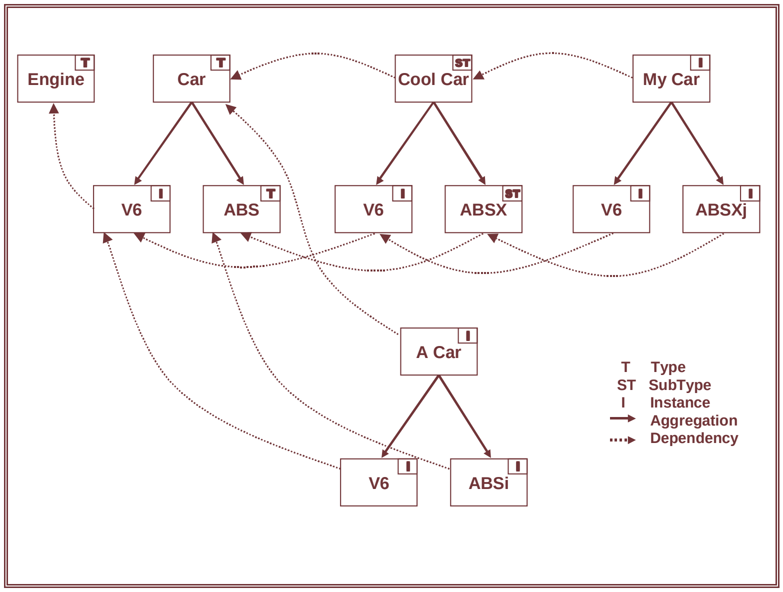
Type Inheritance Concepts

The type inheritance concepts in GME 2000 closely resemble those of object-oriented programming languages. The only significant difference is that in GME, model types are similar in appearance to model instances; they too are graphical, have attributes and contain parts. By default, a model created from scratch is a type. A subtype of a model type can be created by dragging the type and dropping it while pressing the ALT+SHIFT key combination. An instance is created in similar manner, but only the ALT key needs to be used.

A subtype or an instance of a model type depends on the type. There is one significant rule that is different for subtypes and instances. New parts are allowed in a subtype, but not in an instance. Otherwise, parts can be renamed, set membership can be changed, and references can be redirected in both subtypes and instances. Parts cannot be deleted and connections cannot be modified in either subtypes or instances.

Any modification of parts in a type propagates down the inheritance hierarchy. For example, if a part is deleted in a type, the same part will be automatically deleted in all of its instances and subtypes and instances of subtypes all the way down the inheritance hierarchy.

Types can contain other types as well as instances as parts. The mixture of aggregation and type inheritance introduces another kind of relationship between objects. This is best illustrated through an example. In the figure below, there are two root type models: the Engine and the Car. The car contains an instance of an engine, V6, and an ABS type model. V6 is an instance of the Engine; this relationship is indicated by the dash line. Aggregation is depicted by solid lines.

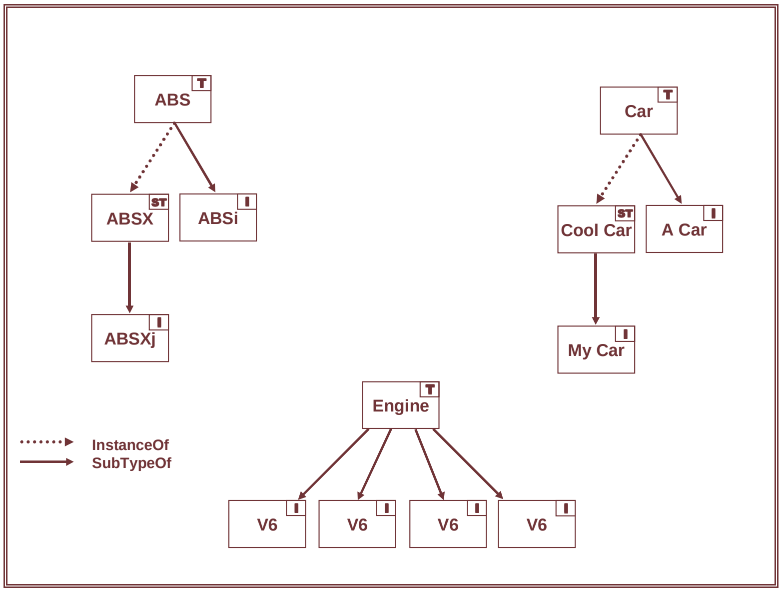


Model Dependency Chains

When we create a subtype of the Car (Cool Car above), we indirectly create another instance of the Engine (V6) and a subtype of the ABS type. This is the expected behavior as a subtype without any modification should look exactly like its base type. Notice the arrow that points from V6 in Cool Car to V6 in Car. Both of these are instances, but there is a dependency between the two objects. If we modify V6 in Car, V6 in Cool Car should also be modified automatically for the same reason: if we don't modify Cool Car it should always look like Car itself. The same logic applies if we create an instance of Cool Car (My Car above). It introduces a dependency (among others) between V6 in My Car and V6 in Cool Car. As the figure shows, this forms a dependency chain from V6 in My Car through V6 in Cool car and V6 in Car all the way to the Engine type model.

What happens if we modify V6 in Cool Car by changing an attribute? Should an attribute change in V6 in Car propagate down to V6 in Cool Car and below? No, that attribute has been overridden and the dependency chain broken with respect to that attribute. However, if the same attribute is changed in V6 in Cool Car, that should propagate down to V6 in My Car unless it has already been overridden there. The same logic applies to preferences.

The figure below shows the same set of models, but only from the pure type inheritance perspective.



Type Inheritance Hierarchy

Let's summarize the rules of type inheritance in GME 2000.

- Parts cannot be deleted in subtypes or instances.
- Parts can be added in subtypes only.
- Part changes in a type model propagate down the type inheritance hierarchy unconditionally.
- Aggregation and type inheritance introduce dependency chains between models.
- Attribute and preference changes, set membership modification and reference redirection propagate down the dependency chain. If a particular setting has been overridden in a certain model in the dependency chain, that breaks the chain for that setting. Changes up in the chain do not propagate to the given model or below.
- The rules for reference redirection are as follows. A null reference in a type can be redirected in any way that the paradigm allows down the dependency chain. A reference to a type in a type model can only be redirected to subtypes or instances of the referred-to type or any instances of any its subtypes. A reference to an instance model in a type model cannot be redirected at all down the hierarchy. Obviously, a reference in an archetype can be redirected in any way the paradigm allows.
- To avoid multiple dependency chains between any two objects, in version 1.1 or older, only root type models could be explicitly derived or instantiated. This restriction has been relaxed. Now, if none of a type model's descendants and ascendants are derived or instantiated, then the model can be derived or instantiated. This means, for example, that a model, that has nor subtypes or instances itself, can contain a model type AND its instances. This relaxed restriction still does not introduce multiple dependency chains.

Attributes and Preferences

The attributes and the preferences dialog boxes each show a checkbox next to each attribute or preference. This box controls the propagation of the settings. If the checkbox is off, the control is read-only and shows the inherited value. If it is on, the user can type in or select the value for the attribute or preference. Turning the box off resets the attribute/preference to the inherited value.

References and Sets

As mentioned before, references can be redirected (with some restrictions) and set membership can be changed in subtypes and instances. The propagation of settings along the dependency chain is true here too. Instead of an explicit checkbox like with attributes and preferences, simply changing the settings breaks the dependency chain for the given object. However, the setting can be easily reset by selecting the Reset item in the appropriate context menu.

References can also be reset to null by using the Clear item in the context menu. However, this is only allowed if the container model is an archetype or if the inherited value of the reference is null itself (otherwise it would violate the rules of inheritance in GME 2000).

Decorators

GME 2000 v1.2 and later implements object drawing in a separate plugable COM module making domain-specific visual representation a reality. In earlier versions of GME one could only specify bitmap files for objects. This method is still supported by the default decorator component shipped with GME 2000.

Replacing the default implementation basically consists of two steps. First we have to create a COM based component, which implements the IMgaDecorator COM interface. Second, we have to assign this decorator to the classes in our metamodel (or for the objects in our model(s) if we want to override the default decorator specified in the metamodel).

GME instantiates a separate decorator for each object in each aspect, so we have to keep our decorator code as compact as possible. Decorator components always have to be in-process servers. Using C++, ATL or MFC is the recommended way to develop decorators.

The IMgaDecorator interface

The following diagram shows the method invocation sequence on the IMgaDecorator interface. Understanding the protocol between GME and the decorators is the key to developing decorators. All the methods on the decorator interface are called by GME (there is no callback mechanism). The direction column in the diagram shows the direction of the information flow.

GME always calls your methods in a read-only MGA transaction. You must not initiate new transactions in your decorator. SaveState() method is the only exception to this rule. This method is called in a read-write transaction, therefore, this is the only place where you can store decorator specific information in the MGA project.

GME	Dir	Decorator
	→	decorator class constructor
	→	GetFeatures(*[out] features*)
	→	SetParam(*[in] name, [in]value*)
	←	GetParam(*[in] name, [out] value*)
	→	Initialize(*[in] mgaproject, [in] mgametapart, [in] mgafco*)
	←	GetPreferredSize(*[out] sizex, [out] sizey*)
	←	GetPorts(*[out] portFCOs*)
	→	SetLocation(*[in] sx, [in] sy, [in] ex, [in] ey*)
	←	GetPortLocation(*[in] fco, [out] sx, [out] sy, [out] ex, [out] ey*)
	←	GetLabelLocation(*[out] sx, [out] sy, [out] ex, [out] ey*)
	←	GetLocation(*[out] sx, [out] sy, [out] ex, [out] ey*)
	→	SetActive(*[in] isActive*)
	→	Draw(*[in] hDC*)
	→	SaveState()
	→	Destroy()

IMgaDecorator Functions

HRESULT GetFeatures([out] feature_code *features)

This method tells GME which features the decorator supports. Available feature codes are (can be combined using the bitwise-OR operator):

F_RESIZABLE : decorator supports resizable objects

F_MOUSEVENTS : decorator handles mouse events

F_HASLABEL : decorator draws labels for objects (outside of the object)

F_HASSTATE : decorator wants to save information in the MGA project

F_HASPORTS : decorator supports ports in objects

F_ANIMATION : decorator expects periodic calls of its draw method

HRESULT SetParam([in] BSTR name, [in] VARIANT value)

If there are some parameters specified for this decorator in the meta model, GME will call this method for each parameter/value pair.

HRESULT GetParam([in] BSTR name, [out] VARIANT *value)

The decorator needs to be able to give back all the parameter/value pairs it got with the SetParam(…) method.

HRESULT Initialize([in] IMgaProject* project, [in] IMgaMetaPart *meta, [in] IMgaFCO *obj)

This is your constructor like function. Read all the relevant data from the project and cache them for later use (it is a better approach than querying the MGA project in

your drawing method all the time). GME will instantiate a new decorator if its MGA object changes.

HRESULT GetPreferredSize([out] long* sizex, [out] long* sizey)

Your decorator can give GME a hint about the size of the object to be drawn. You can compute this information based on the inner structure of the object or based on a bitmap size, or even you can read these values from the registry of the object. However, GME may not take this information into account when it calls your SetLocation() method. All the size and location parameters are in logical units.

HRESULT GetPorts([out, retval] IMgaFCOs **portFCOs)

If your decorator supports ports, it should give back a collection of MGA objects that are drawn as ports inside the decorator. GME uses this method along with successive calls on GetPortLocation() to figure out where can it find port objects.

HRESULT SetLocation([in] long sx, [in] long sy, [in] long ex, [in] long ey)

You have to draw your object exactly to this position in this size. There is no exemption to this. GME always calls this method before Draw().

HRESULT GetPortLocation([in] IMgaFCO *fco, [out] long *sx, [out] long *sy, [out] long *ex, [out] long *ey)

See description of GetPorts(). Position coordinates are relative to the parent object.

HRESULT GetLabelLocation([out] long *sx, [out] long *sy, [out] long *ex, [out] long *ey)

If you support label drawing, you have to specify the location of the textbox of your label. This can reside outside of the object. GME will call SetLocation() before this method.

HRESULT GetLocation([out] long *sx, [out] long *sy, [out] long *ex, [out] long *ey)

Return the coordinates you got in SetLocation().

HRESULT SetActive([in] VARIANT_BOOL isActive)

GME calls this method with VARIANT_FALSE, if your object must be shown in gray color. (Eg.: GME was switched into “set” mode.) By default the decorator should paint its object with the active color.

HRESULT Draw([in] HDC hdc)

You have all the required information when this method is called. Because a Windows HDC is supplied, the decorator has to be an in-process server. Saving and

restoring this DC in the beginning and at the end of your Draw() method is highly recommended.

HRESULT SaveState()

Because this is the only method your decorator is in read-write transaction mode, it has to backup all the permanent data here.

HRESULT Destroy()

A destructor like function. Releasing here all your MGA COM pointers is a good practice.

Using the Decorator skeleton

You can find a decorkit.zip file in the GME 2000 distribution. It contains a skeleton project for Visual C++ that implements a dumb decorator. Modifying the DecoratorConfig.h file would be your first step when using the skeleton.

Assigning decorators to objects

You can assign decorators to objects in your meta model or even later in your model(s). In the MetaGME2000 environment there is a Decorator attribute for each non-connection FCO where you can specify a ProgID along with optional parameter/value pairs for a class. The format of this string is as follows:

`ProgID param1=value1, param2=value2, …`

e.g.:

`MGA.Decorator.MetaDecorator showattributes=false, showabstract=true`

In your models all the non-connection FCOs have a preference setting called Decorator. The format of this string is identical to the one in the meta model.

Metamodeling Environment

The metamodeling environment has been extended with a new decorator component in version 1.2 or later. It displays UML classes including their stereotypes and attributes. Proxies also show this information It resizes UML classes accordingly. Note that the figures below show the old appearance of metamodels.

Version 1.2 adds a new MCL syntax checker add-on to the metamodeling environment. Every time a constraint expression attribute is changed, this add-on is activated. Note that the target paradigm information is not available to this tool, therefore, it cannot check arguments and parameters, such as kindname. These can only be checked at constraint evaluation time in your target environment.

Step by step guide to basic metamodeling

The following sections describe the concepts that are used to model the output Paradigm.

Paradigm

The Paradigm is represented as the model that contains the UML class diagram. The name of the Paradigm model is the name of the paradigm produced by the interpreter. The attributes of the Paradigm are **Author Information** and **Version Information**.

Folder

A Folder is represented as a UML class of stereotype «folder». Folders may own other Folders, FCO’s, and Constraints. Once a Folder contains another container, it by default contains all FCO’s, Folders, and Constraints that are in that container. Folders are visualized only in the model browser window of GME 2000, and therefore do not use aspects. A Folder has the **Displayed Name**, and **In Root Folder** attributes.

How to specify containment for a Folder

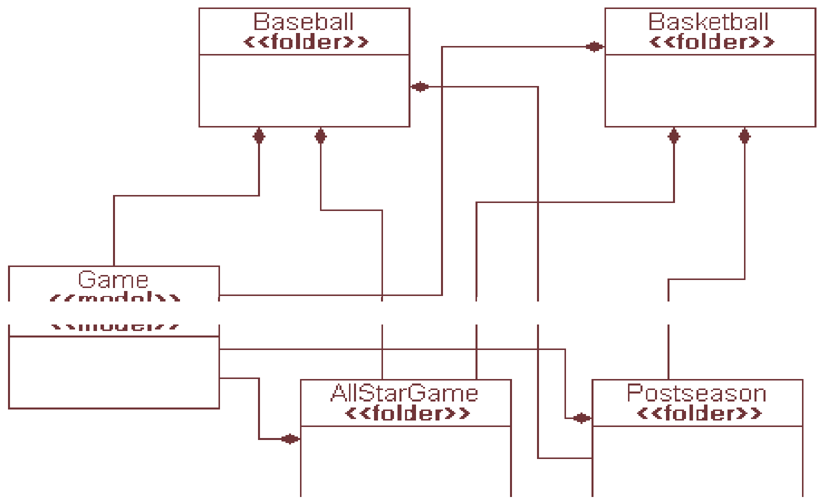
Folder containment applies to Folders and Models that may be contained in a Folder.

In the figure below, the UML diagram outlines the containment scheme of a paradigm for a sports season. To specify containment for a Folder, follow these steps.

Create the **Folder** and **item** it contains (through insertion, or dragging from the parts menu)

Connect the **item** to the **Folder**

Now, the Folder contains the item.



Example of a Folder containment

FCO

This is a class that is mandatorily abstract. The purpose of this class is to enable objects that are inherently different (Atom, Reference, Set, etc.) to be able to inherit from a common base class.

To avoid confusion with the generalization of modeling concepts (Model, Atom, Set, Connection, Reference) called collectively an "FCO", and this *kind* of object in the metamodeling environment which is called an "FCO", the metamodeling concept (that would actually be dragged into a Paradigm model) will be shown in regular font, while the generalization of types will be in italics as *FCO*. An FCO has the **Is Abstract** and **General Preferences** attributes. All *FCO-s* will also have these attributes.

How to create an FCO

An FCO (like all *FCO-s*) is created by dragging in the atom corresponding to its stereotype, or inserting the atom through the menu.

How to specify an Attribute for an FCO

Create and configure the **Attribute** and the **FCO.**

Connect the **Attribute** to the **FCO**

Now, the Attribute belongs to the FCO.

Atom

This class represents an Atom. The Atom is the simplest kind of object in one sense, because it cannot contain any other parts; but it is complex to define because of the many different contributions it can make to a Model, Reference, etc.

An Atom has the **Icon Name**, **Port Icon Name**, and **Name Position** attributes.

How to set that an Atom is a Port

Configure the **Atom** to be a member of a **Model**

Click on the attributes of the Containment association between the **Atom** and the **Model**

Assert the **Object Is A Port** attribute.

Reference

To represent a Reference class, two things must be specified: the FCO to which this Reference refers, and the Model to which the Reference belongs. A Reference has the **Icon Name** and **Name Position** attributes.

How to specify containment of a Reference in a Model

Connect the **Reference** to the **Model**

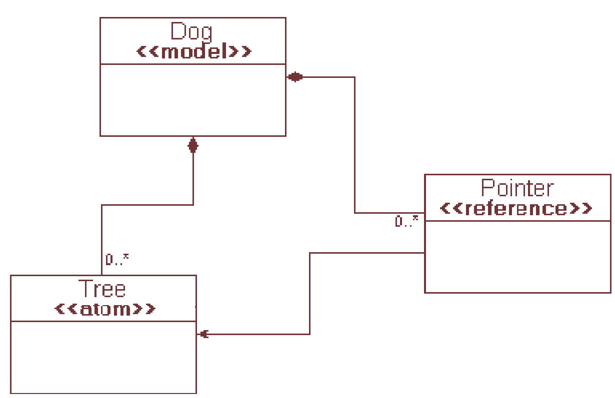
Resolve the prompt for connection type as "**Containment**".

How to specify the FCO to which a Reference refers

Connect the Reference to the FCO.

If the FCO is of type Model, an additional prompt is displayed (exactly the same as when giving ownership to the Model as in the previous step). This time, choose the "Refer" type of connection. If the FCO is not of type Model, then no additional input is necessary.

When specifying the roles to which a Reference may refer (that is, if the referred FCO may play more than one kind of role in a particular Model), the current solution is that it may refer to all roles of that particular kind. However, in the future, this list may be modified during paradigm construction through the help of an add-on.



Example implementation of a Reference.

Connection

In order for a Connection to be legal within a Model, it must be contained through aggregation in that Model. The Connection is another highly configurable concept. The attributes of a Connection include **Name Position**, **1st destination label**, **2nd destination label**, **1st source label**, **2nd source label**, **Color**, **Line type**, **Line end**, and **Line Start**.

How to specify a connection between two Atoms

In addition to Atoms, a Reference to an Atom may also be used as an endpoint of the Connection. Note that Connection is also usable as an endpoint, but there is currently no visualization for this concept.

Drag in a **Connector** Atom (the name of the Connector was deleted in the example figure)

Connect the source **Atom** to the **Connector**

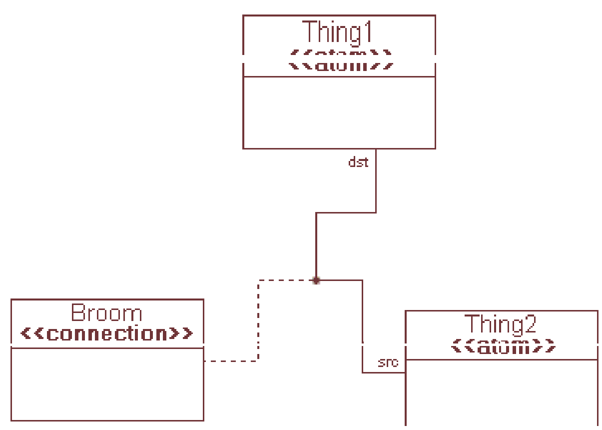
Connect the **Connector** to the destination **Atom**

Connect the **Connector** to the **Connection**. Resolve the Connection type to "**AssociationClass**"

The rolenames of the connections ("src" and "dst") denote which of the Atoms may participate as the source or destination of the connection. There may be only one source and one destination connection to the Connector Atom.

Inheritance is a useful method to increase the number of sources and destinations, since all child classes will also be sources and destinations.

Currently, all possible FCO source/destination combinations will be used in the production of the metamodel. However, in future revisions of the metamodeling environment, the list of allowable connections may be modified at model building time (to eliminate certain possibilities from ever occurring).

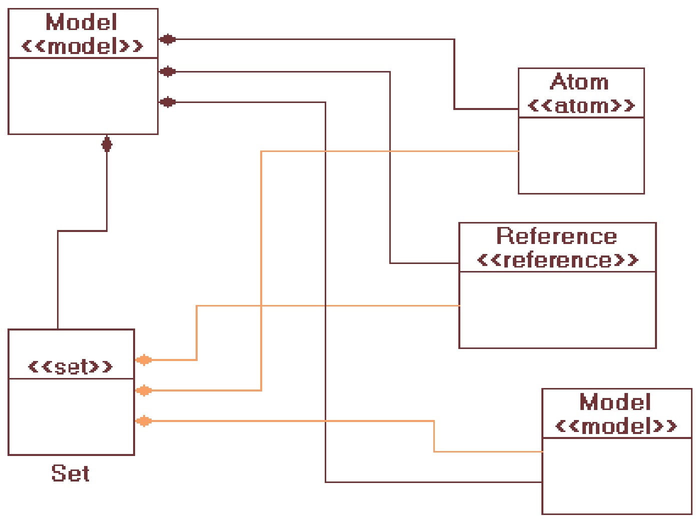


Example of a Connection

Set

The Set is a more general case of the Reference. Sets have the **Icon name**, and **Name Position** attributes.

Figure 4 shows an example implementation of a Set. The members of the Set are "owned" by the Set through the "SetMembership" connection kind (when connecting the Reference to the Set, the user will be prompted to choose between the "SetMembership" and "ReferTo" connection kinds). Some underlying assumptions exist here, such as all members of the Set must be members of the Model to which this set belongs.



Example implementation of a Set

How to specify what FCO-s *a Set "Owns"*

Connect the ***FCO*** to the **Set** Atom. In the event of an ambiguity, resolve it with the **SetMembership** connection type.

Make sure to aggregate the **Set** to the **Model** in which it will reside.

Model

The Model may contain (through the "Containment" connection type) any other FCO, and it associates a role name to each FCO it contains. The Model has the **Name Position** and **In Root Folder** attributes.

How to contain a Model (Model-1) in a Model (Model-0)

Connect Model-1 to Model-0

Note that it is applicable to have a Model contain itself (the previous case where Model-1 == Model-0).

How to contain an Atom in a Model

In the event that an FCO is used as a superclass for the Model, then FCO may replace Model in the following sequence. Atom may be replaced by Set, Reference, or Connection.

Create and configure the **Atom** and the **Model**

Connect the **Atom** to the **Model**

Attributes

Attributes are represented by UML classes in the GME metamodeling environment. There are three different kinds of Attributes: Enumerated, Field, and Boolean. Once any of these Attributes are created, they are aggregated to *FCO-s* in the Attributes Aspect. The order of attributes an FCO will have is determined by the relative vertical location of the UML classes representing the attributes.

Inheritance

Inheritance is standard style for UML. Any *FCO* may inherit from an FCO kind of class, but an FCO may inherit only from other FCO's. Kinds may inherit only from each other (e.g. Model may not inherit from Atom). When the class is declared as abstract, then it is used during generation, but no output FCO is generated. No class of kind FCO is ever generated.

When multiple-inheritance is encountered, it will always be treated as if it were virtual inheritance. For example, the classic diamond hierarchy will result in only one grandparent class being created, rather than duplicate classes for each parent.

How to Specify Inheritance

It is assumed that Child and Parent are of the same kind (e.g. Atom, Model). FCO is used in this example, for brevity, but note that any *FCO* may participate in the Child role, if the Parent is of kind FCO. Else, they must match.

Connect the Parent **FCO** to the **Inheritance** Atom. This creates a superclass.

Connect the **Inheritance** atom to the Child **FCO**. This creates the child class.

Aspect

This set defines the visualization that the Models in the destination paradigm will use. Models may contain Aspects through the "HasAspect" connection kind. This is visualized using the traditional UML composition relation using a filled diamond. FCOs that need to be shown in the an aspect must be made members of the given Aspect set.

GME 2000 supports aspect mapping providing precise control over what aspect of a model is shown in an aspect of the containing model. This is advanced rarely-used usually feature is typically applied in case a container and a contained models have disjoint aspect sets. Specifying aspect mapping would be to cumbersome in a UML-like graphical language. The metamodeling interpreter allows specifying this information in a dialog box (described in detail later).

Constraints

Constraints may be specified as owned by a particular kind of FCO. This means that there is a certain amount of scope granted to the checking of the constraint. Any constraint not owned by a Folder or FCO is attributed to the paradigm.

Constraints are aggregated to *FCO-s* and Folders in the Constraints Aspect.

Note that *constraint functions* are also supported. These represent MCL constraint equations that can be reused in constraint expressions.

Composing Metamodels

The new composable metamodeling environment released with GME 2000 v1.1, supports metamodel composition. First, it supports multiple paradigm sheets. Unlike most UML editors, where boxes representing classes are tied together by name, GME 2000 uses references. They are called proxies. Any UML class atom can have multiple proxies referring to it. These references are visualized by a curved arrow inside the regular UML class icon. The atom and all its proxies represent the same UML class.

New operators

In addition to improving the usability of the environment and the readability of the metamodels, the primary motivation behind composable metamodeling is to support the reuse of existing metamodels and, eventually, to create extensive metamodel libraries. However, this mandates that existing metamodels remain intact in the composition, so that changes can propagate to the metamodels where they are used.

The above requirement and limitations of UML made it necessary to develop three new operators for use in combining metamodels together: an equivalence operator, an implementation inheritance operator, and an interface inheritance operator.

Equivalence operator

The equivalence operator is used to represent the (full) union between two UML class objects. The two classes cease to be two separate classes, but form a single class instead. Thus, the union includes all attributes and associations, including generalization, specialization, and containment, of each individual class. Equivalence can be thought of as defining the "join points" or "composition points" of two or more source metamodels.

Implementation inheritance operator

The semantics of UML specialization (i.e. inheritance) are straightforward: specialized (i.e. child) classes contain all the attributes of the general (parent) class, and can participate in any association the parent can participate in. However, during metamodel composition, there are cases where finer-grained control over the inheritance operation is necessary. Therefore, we have introduced two new types of inheritance operations between class objects—implementation inheritance and interface inheritance.

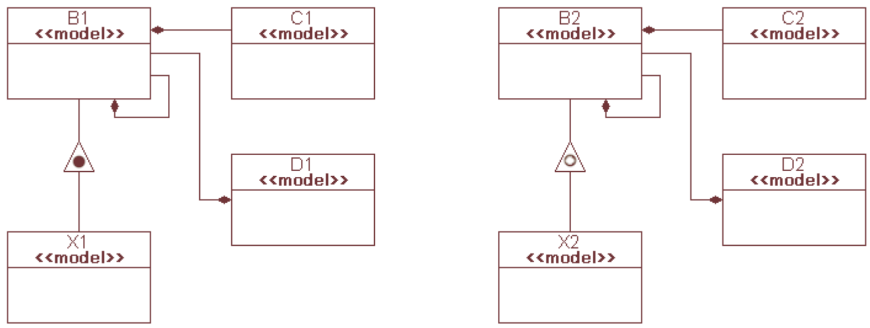
In implementation inheritance, the subclass inherits all of the base class' attributes, but only those containment associations where the base class functions as the container. No other associations are inherited. Implementation inheritance is represented graphically by a UML inheritance icon containing a solid black dot.

This can be seen in the left hand side diagram in the figure below, where implementation inheritance is used to derive class X1 from class B1. In this case, X1 the association allowing objects of type C1 to be contained in objects of type B1. In other words, X1-type objects can contain C1-type objects. Because B1-type objects can contain other B1-type objects, X1-type objects can contain objects of type B1 but not of type X1. Note that D1-type objects can contain objects of type B1 but not objects of type X1.

Interface inheritance operator

The right side of the figure shows interface inheritance between B2 and X2 (the unfilled circle inside the inheritance icon denotes interface inheritance). Interface inheritance allows no attribute inheritance but does allow full association inheritance, with one exception: containment associations where the base class functions as the container are not inherited. Therefore, in this example, X2-type objects can be contained in objects of type D2 and B2, but no objects can be contained in X2-type objects, not even other X2-type objects.

The union of implementation inheritance and interface inheritance is the normal UML inheritance. It should also be noted that these operators could have been implemented using UML stereotypes. However, interface and implementation inheritance are semantically much closer to regular inheritance than to associations. Therefore, the use of association with stereotypes would be misleading.



Implementation and interface inheritance operators

Aspect equivalence

Since classes representing Aspects show up only in the Visualization aspect, another new operator is used to express the equivalence of aspects, called the SameAspect operator. While aspects can have proxies as well, they are not sets any more; they are references. Hence, they cannot be used to add additional objects to the aspect. In this case, a new aspect needs to be created. New members can be added to it, since it is a set. Using the SameAspect operator and typically a proxy of another aspect, the equivalence of the two aspects can be expressed.

Note that having two aspects with the same name without explicitly expressing the equivalence of them will result in two different aspect in the target modeling paradigm.

The name of the final aspect is determined by the following rules. If an equivalence is expressed between a proxy and a UML class, the name of the class is used. If one of them is abstract and the other is not, the name of the non-abstract class (or proxy) is used. If both aspects are proxies (or classes), then the name of the SameAspect operator is used.

Currently, the order of aspects in the target paradigm is determined by the relative vertical position of the aspect set icons in the metamodels.

Folder equivalence

The equivalence of folders can be expressed using the SameFolder operator.

Generating the Target Modeling Paradigm

Once the Paradigm Model is complete, then comes time to interpret the Model. Interpretation can be initiated from any model. After extensive consistency checking, the interpreter displays a dialog box where aspect mapping information can be specified.

Aspect Mapping

The dialog box contains as many tabs as there are distinct aspects in the target environment. Under each tab a listbox displays all possible model-role combinations in the first column. The second column presents the available aspects for the given model and model reference (i.e. in the specified role) in a combo box. The default selection is the aspect with the same name as the container models aspect. For all other FCOs (atoms, sets, connections) this files shows N/A.

The third column is used to specify whether the given the aspect is primary or not for the given FCO (i.e. in the specified role). In a primary aspect, the given FCO can be added or deleted. In a secondary aspect, it only shows up, but cannot be added or deleted.

Note that all the information provided by the user through this dialog box is persistent. It is stored in the metamodel, in the registry of the corresponding objects. A subsequent invocation of the interpreter will show the dialog box with the information specified by the user the previous time.

Attribute Guide

Each attribute of any given *FCO* in the Metamodeling environment has a specific meaning for the output paradigm. This section describes each attribute, and lists the *FCO(s)* in which the attribute resides. Attributes are listed by the text prompted on the screen for their entry. The section also gives what special instructions (if any) are necessary for filling out the attribute.

For fields, if the default value of the field is "", then no default value is specified in the description. All other attributes list the default value.

1st source label

String value that gives the *name* of the Attribute class to be displayed there. The Attribute should also belong (through aggregation) to the Connection. Then, the value of that Attribute will be displayed in the first position at the end of the source of the connection.

Contained in – **Connection**

2nd source label

String value that gives the *name* of the Attribute class to be displayed there. The Attribute should also belong (through aggregation) to the Connection. Then, the value of that Attribute will be displayed in the second position at the end of the source of the connection.

Contained in – **Connection**

1st destination label

String value that gives the *name* of the Attribute class to be displayed there. The Attribute should also belong (through aggregation) to the Connection. Then, the value of that Attribute will be displayed in the first position at the end of the destination of the connection.

Contained in – **Connection**

2nd destination label

String value that gives the *name* of the Attribute class to be displayed there. The Attribute should also belong (through aggregation) to the Connection. Then, the value of that Attribute will be displayed in the second position at the end of the destination of the connection.

Contained in – **Connection**

Abstract

Boolean checkbox that determines whether or not the FCO in question will actually be generated in the output paradigm. If the checkbox is checked, then no object will be created, but all properties of the FCO will be passed down to its inherited children (if any).

Default value – **Unchecked**

Contained in – **FCO, Atom, Model, Set, Connection, Reference**

Author information

A text field translated into a comment within the paradigm output file.

Contained in – **Paradigm**

Cardinality

Text field that gives the cardinality rules of containment for an aggregation.

Default value – **0..***

Contained in – **Containment, FolderContainment**

Color

String value that gives the default color value of the connection (specified in hex, ex: 0xFF0000).

Default value – **0x000000** (black)

Contained in – **Connection**

Composition role

Text field that gives the rolename that the FCO will have within the Model.

Contained in – **Containment**

Constraint Equation

Multiline text field that gives the MCL equation for the constraint.

Contained in – **Constraint**

Data type

Enumeration that gives the default data type of a FieldAttr. The possible values are String, Integer, and Double.

Default value – **String**

Contained in – **FieldAttr**

Decorator

Test field that specifies the decorator component to be used to display the given object in the target environment. Example: MGA.Decorator.MetaDecorator

Contained in – Model, Atom, Reference, Set

Default = 'True'

A boolean checkbox that describes the default value of a BooleanAttr.

Default value – **Unchecked**

Contained in – **BooleanAttr**

Default parameters

Text field that gives the default parameters of the constraint.

Contained in – **Constraint**

Default menu item

Text field that gives the *displayed name* of the menu item in the **Menu items** attribute to be used as the default value of the menu.

Contained in – **EnumAttr**

Description

Text field that is displayed when the constraint is violated.

Contained in – **Constraint**

Displayed name

String value that gives the displayed name of a Folder or Aspect. This will be the value that is shown in the model browser, or aspect tab (respectively). A blank value will result in the displayed name being equal to the name of the class.

Contained in – **Folder, Aspect**

Field default

Text field that gives the default value of the FieldAttr.

Contained in – **FieldAttr**

General preferences

Text field (multiple lines) that allows a user to enter data to be transferred directly into the XML file. This is a highly specific text area, and is normally not used. The occasions for using this area is to configure portions of the paradigm that the Metamodeling environment has not yet been developed to configure.

Contained in – **FCO, Atom, Model, Set, Connection, Reference**

Global scope

A boolean checkbox that refers to the definition scope of the attribute. In most cases, it is sufficient to leave this attribute in its default state (true). The reason for giving the option of scope is to be able to include attributes with the same names in different *FCO-s*, and have those attributes be different. In this case, it is necessary to include local scoping (i.e. remove the global scope), or the paradigm file will be ambiguous.

Default value – **Checked**

Contained in – **EnumAttr, BooleanAttr, FieldAttr**

Icon

Text field that gives the name of a file to be displayed as the icon for this object.

Contained in – **Atom, Set, Reference, Model**

In root folder

Boolean checkbox that determines whether or not this object can belong in the root folder. Note that if an object cannot belong to the root folder, then it must belong to a Folder or Model (somewhere in its containment hierarchy) that *can* belong to the root folder.

Default value – **Checked**

Contained in – **Folder, Model, Atom, Set, Reference**

Line end

Enumeration of the possible end types of a line. Possible types are Butt (no special end), Arrow, and Diamond.

Default value – **Butt**

Contained in – **Connection**

Line start

Enumeration of the possible start types of a line. Possible types are Butt (no special end), Arrow, and Diamond.

Default value – **Butt**

Contained in – **Connection**

Line type

Enumeration of the possible types of a line. Possible types are Solid, and Dash.

Default value – **Solid**

Contained in – **Connection**

Number of lines

Integer field that gives the number of lines to display for this FieldAttr.

Default value – **1**

Contained in – **FieldAttr**

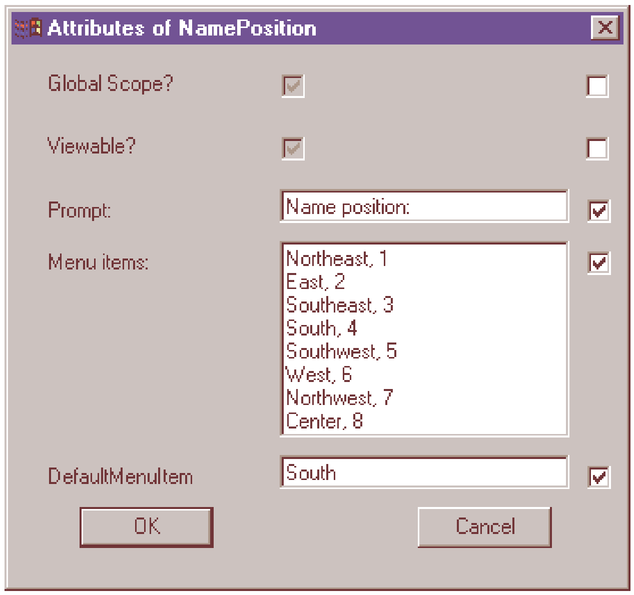
Menu items

A text field that lists the items in an EnumAttr. There are two modes for this text field (which can also be called a text box, because it has the ability for multiple lines).

In basic mode, the field items are separated by carriage returns, in the order in which they should be listed in the menu. In this case, the text used as the menu will be the same as value of the menu.

In the expanded mode, it is possible to list the definite values to be used for the menu elements. This is done by separating the displayed value from the actual value with a comma (,).

Example:



Sample enumerated attribute specification

Note that the displayed and actual value need not be of the same basic type (character, integer, float, etc.) because it will all be converted to text.

Contained in – **EnumAttr**

Name position

Enumeration that lists the nine places that the name of an FCO can be displayed.

Default value – **South**

Contained in – **Atom, Set, Reference, Model**

Object is a port

Boolean checkbox that determines whether or not the FCO will be viewable as a port within the model.

Default value – **Unchecked**

Contained in – **Containment**

On…

The Constraint has many attributes which are similar, except for the type of event to which they refer. They are all boolean checkboxes that give the constraint manager the authority to check this constraint when certain events occur (e.g. Model creation/deletion, connecting two objects). For more information on the semantics of these events, please refer to the constraint manager documentation.

- On close model
- On new child
- On delete
- On disconnect
- On connect
- On derive
- On change property
- On change assoc.
- On exclude from set
- On include in set
- On move
- On create
- On change attribute
- On lost child
- On refer
- On unrefer

Default value – **Unchecked**

Contained in – **Constraint**

Port icon

Text field that gives the name of a file to be displayed as the port icon for this object. If no entry is made for this field, but the object is a port, then the normal icon will be scaled to port size.

Contained in – **Atom, Set, Reference, Model**

Priority (1=High)

Enumeration of the possible levels of priority of this constraint. For more information on constraint priority, refer to the constraint manager.

Contained in – **Constraint**

Prompt

A text field translated into the prompt of an attribute. It is in exact WYSIWYG format (i.e. no ':' or '-' is appended to the end).

Contained in – **EnumAttr, BooleanAttr, FieldAttr**

Rolename

Text field that gives the rolename that the FCO will have in the Connection. There are two different possible default values, 'src' and 'dst', depending upon whether the connection was made from the Connector to the FCO, or the FCO to the Connector.

Default value – **src** or **dst**

Contained in – **SourceToConnector, ConnectorToSource**

Version information

A text field translated into a comment within the paradigm output file. The user is responsible for updating this field.

Contained in – **Paradigm**

Viewable

A boolean checkbox that decides whether or not to display the attribute in the paradigm. If the state is unchecked, then the attribute will be defined in the metamodel, but not viewable in any Aspect (regardless of the properties of the *FCO*. This is useful if you want to store attributes outside the user's knowledge.

Default value – **Checked**

Contained in – **EnumAttr, BooleanAttr, FieldAttr**

Semantics Guide to Metamodeling

The following table displays the representation of the concepts of GME 2000, and how they translate semantically into core MGA concepts.

Stereotype/name	Context	Semantics [& Implications]
First Class Objects (FCO's)		
«model»	A class	The class is an MGA model
«atom»	A class	The class is an MGA atom
«connection»	A class	The class is an MGA connection (must be used as an Association Class
«reference»	A class	The class is an MGA reference
«set»	A class	The class is an MGA set
«FCO»	A class (abstract only)	The class is a base type of another FCO
Associations		
Containment	An association (with diamond) between a «model» and an FCO	The «model» contains the specified FCO as a part.
AssociationClass	An association between a «connection» (class) and an Association Connector (models the connection join).	The «connection» contains all of the roles that the Association Connection has.
ReferTo	A directed association between a «reference» and a «model», «atom», or «reference»	The instances of the «reference» class will refer to the instances of the «model», «atom», or «reference» class.
Association Classes		
«connection»	An association between a src/dst pair (or an n-ary connection, in the general sense) that is attributed by a «connection» class	The «connection» class represents the src/dst pair(s) as an MGA connection. [note: the «connection» is an FCO]
Containment		
FolderContainment	An association (with diamond) between a «folder» and a «folder»	The «folder» contains 0..n of the associated «folder» as a legal sub-folder
FolderContainment	An association (with diamond) between a «folder» and an FCO	The «folder» contains 0..n of the associated FCO as a legal root-object
Containment	An association (with diamond) between a «model» and an FCO	The «model» contains the associated FCO which plays a specified role
SetMembership	An association (with diamond) between a «set» and an FCO	The «set» may contain the associated FCO.
HasAspect	An association between a «model» and an «aspect»	The «model» contains the specified «aspect».
Cardinality		
(none)	An integer attribute for each end of the association	This end of the association has the cardinality specified [unspecified cardinality is assumed to be 1]
Various		
«aspect»	A class	The class denotes an MGA aspect
«folder»	A class	The class denotes an MGA folder
(none)	The model represents a Project	An MGA Project
Inheritance		
(none)	UML Inheritance	The class inherits from a superclass. An attribute of the destination is the rolename to be used for the child class.
Groups of parts		
Connector	Atom, reference, (port), (reference port)	The part may play a role in a connection
FCO	Model, atom, reference, connection, set	The part is a first class object
Referenceable	Model, atom, reference	The part may be referenced

High-Level Component Interface

Introduction to the Component Interface

The process of accessing GME 2000 models and generating useful information, e.g. configuration files for COTS software, database schema, input for a discrete-event simulator, or even source code, is called ***model interpretation***. GME provides two interfaces to support model interpretation. The first one is a COM interface that lets the user write these components in any language that supports COM, e.g. C++, Visual Basic or Java. The COM interface provides the means to access and modify the models, their attributes and connectivity. In short, the user can do everything that can be done using the GUI of the GME. There is a higher-level C++ interface that takes care of a lot of lower level issues and makes component writing much easier. This high-level C++ component interface is the focus of this chapter.

Interpreters are typical, but not the only components that can be created using this technology. The other types are ***plugins***, i.e. components that provide some useful additional functionality to ease working in GME. These components are very similar to interpreters, though they are paradigm-independent. For example, a plugin can be developed to search or locate objects based on some user defined criteria, like the value of an attribute.

What Does the Component Interface Do?

The component interface is implemented on the top of the COM interface. When the user initiates model interpretation, the component interface creates the so-called Builder Object Network (BON). The builder object network mirrors the structure of the models: each model, atom, reference, connection, etc. has a corresponding builder object. This way the interface shields the user from the lower level details of the COM interface and provides support for easy traversal of the models along either the containment hierarchy, the connections, or the references. The builder classes provide general-purpose functionality. The builder objects are instances of these predefined paradigm independent classes. For simple paradigm-specific or any kind of paradigm independent components, they are all the user needs. For more complicated components, the builder classes can be extended with inheritance. By using a pair of supplied macros, the user can have the component interface instantiate these paradigm-specific classes instead of the built-in ones. The builder object network will have the functionality provided by the general-purpose interface extended by the functionality the component writer needs.

Component Interface Entry Point

The Builder2000.h file in component source package defines the high-level C++ component interface. The entry point of the component is defined in the Component.h in the appropriate subdirectory of the components directory. Here is the file at the start of the component writing process:

```
#ifndef GME_INTERPRETER_H
#define GME_INTERPRETER_H

#include "Builder2000.h"

#define NEW_BON_INVOKE
//#define DEPRECATED_BON_INVOKE_IMPLEMENTED

class CComponent {
public:
  CComponent() : focusfolder(NULL) { ; }
  CBuilderFolder *focusfolder;
  CBuilderFolderList selectedfolders;
  void InvokeEx(CBuilder &builder,CBuilderObject *focus,
       CBuilderObjectList &selected, long param);
//void Invoke(CBuilder &builder,
       CBuilderObjectList &selected, long param);
};

#endif // whole file
```

Before GME 2000 version 1.2 this used to be simpler, but not as powerful. The Invoke function of the CComponent class used to be the entry point of the component. When the user initiates interpretation, first the builder object network is created then the above function is called. The first two parameters provide two ways of traversing the builder object network. The user can access the list of folders through the CBuilder instance. Each folder provides a list of builder objects corresponding to the root models and subfolders. Any builder can then be access through recursive traversal of the children of model builders.

The CBuilderModelList contains the builders corresponding to the models selected at the time interpretation was started. If the component was started through the main window (either through the toolbar or the File menu) then the list contains one model builder, the one corresponding to the active window. If the interpretation was started through a context menu (i.e. right click) then the list contains items for all the selected objects in the given window. If the interpretation was started through the context menu of the Model Browser, then the list contains the builders for the selected models in the browser.

Using this list parameter of the Invoke function makes it possible to start the interpretation at models the user selects. The long parameter is unused at this point.

In version 1.2, Invoke has been replaced by InvokeEx, which clearly separates the focus object from the selected objects. (Depending on the invocation method both of these parameters may be empty.) To maintain compatibility with existing components, the following preprocessor constants have been designated for inclusion in the Component.h file:

- NEW_BON_INVOKE: if #defined in Component.h, indicates that the new BON is being used. If it is not defined (e.g. if the Component.h from an old BON is being used) the framework works in compatibility mode.

- DEPRECATED_BON_INVOKE_IMPLEMENTED: In most cases, only the new CComponent::InvokeEx needs to be implemented by the component programmer, and the ImgaComponent::Invoke() method of the original COM interface also results in a call to InvokeEx. If, however the user prefers to leave the existing Component::Invoke() method to be called in this case, the #define of this constant enables this mode. InvokeEx() must be implemented anyway (as NEW_BON_INVOKE is still defined).

- IMPLEMENT_OLD_INTERFACE_ONLY: this constant can be included in old Component.h files only to fully disable support for the new IMgaComponentEx COM interface (GME invokes to the old interface if the new one is not supported). Using this constant is generally not recommended.

If none of the above constants are defined, the BON framework interface is compatible with the old Ccomponent classes. Censequently, older BON code (Component.h and Component.cpp) can replace the corresponding skeleton/example files provided in the new BON. When using such a component, however, a warning is message is displayed to remind users to upgrade the component code to one fully compliant with the new BON. Although it is strongly recommended to update the component code (i.e converting CComponent::Invoke to CComponent::InvokeEx(), this warning can also be supressed by disabling the new COM component interface through the inclusion of the #define IMPLEMENT_OLD_INTERFACE_ONLY definition into the old Component.h file.

Plung-Ins are paradigm-independent components. The example Noname plug-in displays a message. The implementation is in the component.cpp file shown below:

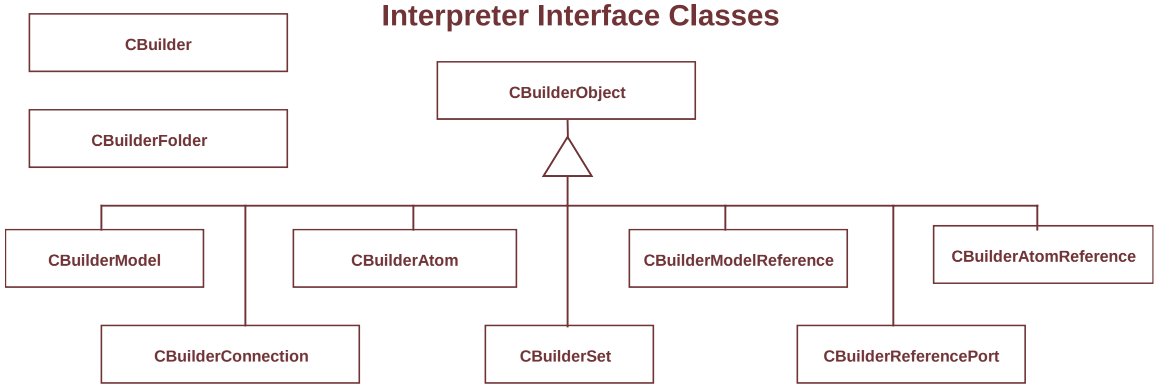
```
#include "stdafx.h"
#include "Component.h"

void CComponent:: InvokeEx(CBuilder &builder,CBuilderObject *focus,
       CBuilderObjectList &selected, long param)
{
  AfxMessageBox("Plug-In Sample");
}
```

The component.h and component.cpp files are the ones that the component writer needs to expand to implement the desired functionality.

Component Interface

The simple class structure of the component interface is shown below. Note that each class is a derivative of the standard MFC CObject class.

Interpreter Interface Classes



As noted before, the single instance of the CBuilder class provides a top level entry point into the builder object network. It provides access to the model folders and supplies the name of the current project. The public interface of the CBuilder class is shown below.

```
class CBuilder : public CObject {
public:
  CBuilderFolder *GetRootFolder() const;
  const CBuilderFolderList *GetFolders() const;
  CBuilderFolder *GetFolder(CString &name) const;
  CString GetProjectName() const;
};
```

The CBuilderFolder class provides access to the root models of the given folder. It can also be used to create new root models.

```
class CBuilderFolder : public CObject {
public:
  const CString& GetName() const;
  const CBuilderModelList *GetRootModels() const;
  const CBuilderFolderList *GetSubFolders() const
  CBuilderModel *GetRootModel(CString &name) const;
  CBuilderModel *CreateNewModel(CString kindName);
};
```

The CBuilderObject is the base class for several other classes. It provides a set of common functionality for models, atoms, references, sets and connections. Some of the functions need some explanation.

The GetAttribute() functions return true when their successfully retrieved the value of attribute whose name was supplied in the name argument. If the type of the val argument does not match the attribute or the wrong name was provided, the function return false. For field and page attributes, the type matches that of specified in the meta, for menus, it is a CString and for toggle switches, it is a bool.

The GetxxxAttributeNames functions return the list of names of attributes the given object has. This helps writing paradigm-independent components (plug-ins).

The GetReferencedBy function returns the list of references that refer to the given object (renamed in v1.2 from GetReferences).

The GetInConnections (GetOutConnection) functions return the list of incoming (outgoing) connections from the given object. The string argument specifies the name of the connection kind as specified by the modeling paradigm. The GetInConnectedObjects (GetOutConnectedObjects) functions return a list of objects instead. The GetDirectInConnections (GetDirectOutConnections) build a tree. The root of the tree is the given object, the edges of the tree are the given kind of connections. The function returns the leaf nodes. Basically these functions find paths to (from) the given object without the component writer having to write the traversal code.

The TraverseChildren virtual functions provide a ways to traverse the builder object network along the containment hierarchy. The implementation provided does not do anything, the component writer can override it to implement the necessary functionality. As we'll see later, the CBuilderModel class does override this function. It enumerates all of its children and calls their Traverse method.

```
class CBuilderObject : public CObject {
  const CString& GetName();
  const bool SetName(CString newname);

  void GetNamePath(CString &namePath) const;

  const CString& GetKindName() const;
  const CString& GetPartName() const;

  const CBuilderModel *GetParent() const;

  CBuilderFolder* GetFolder() const;

  bool GetLocation(CString &aspectName,CRect &loc);
  bool SetLocation(CString aspectName,CPoint loc);

  void DisplayError(CString &msg) const;
  void DisplayError(char *msg) const;
  void DisplayWarning(CString &msg) const;
  void DisplayWarning(char *msg) const;

  bool GetAttribute(CString &name,CString &val) const;
  bool GetAttribute(char *name,CString &val) const;
  bool GetAttribute(CString &name,int &val) const;
  bool GetAttribute(char *name,int &val) const;
  bool GetAttribute(CString &name,bool &val) const;
  bool GetAttribute(char *name,bool &val) const;

  bool SetAttribute(CString &name, CString &val);
  bool SetAttribute(CString &name, int val);
  bool SetAttribute(CString &name, bool val);

  void GetStrAttributeNames(CStringList &list) const;
  void GetIntAttributeNames(CStringList &list) const;
  void GetBoolAttributeNames(CStringList &list) const;

  void GetReferencedBy(CBuilderObjectList &list) const;

  const CBuilderConnectionList *GetInConnections(CString &name) const;
  const CBuilderConnectionList *GetInConnections(char *name) const;
  const CBuilderConnectionList *GetOutConnections(CString &name)const;
  const CBuilderConnectionList *GetOutConnections(char *name) const;

  bool GetInConnectedObjects(const CString &name,
                             CBuilderObjectList &list);
  bool GetInConnectedObjects(const char *name,
                             CBuilderObjectList &list);
  bool GetOutConnectedObjects(const CString &name,
```

```
                               CBuilderObjectList &list);
  bool GetOutConnectedObjects(const char *name,
                               CBuilderObjectList &list);

  bool GetDirectInConnections(CString &name,
                               CBuilderObjectList &list);
  bool GetDirectInConnections(char *name,
                               CBuilderObjectList &list);
  bool GetDirectOutConnections(CString &name,
                               CBuilderObjectList &list);
  bool GetDirectOutConnections(char *name,
                               CBuilderObjectList &list);

  virtual void TraverseChildren(void *pointer = 0);
};
```

The CBuilderModel class is the most important class in the component interface, simply because models are the central objects in the GME. They contain other objects, connections, sets, they have aspects etc. The GetChildren function returns a list of all children, i.e. all objects the model contains (models, atoms, sets, references and connections). The GetModels method returns the list of contained models. If a role name is supplied then only the specified part list is returned. The GetAtoms, GetAtomReferences and GetModelReferences, GetSets() functions work the same way except that a part name must be supplied to them. The GetConnections method return the list of the kind of connections that was requested. These are the connections that are visible inside the given model.

The GetAspectNames function return the list of names of aspects the current model has. This helps in writing paradigm-independent components.

Children can be created with the appropriate creation functions. Similarly, connections can be constructed by specifying their kind and the source and destination objects. Please, see the description of the CBuilderConnection class for a detailed description of connections.

The TraverseModels function is similar to the TraverseChildren but it only traverses models.

```
class CBuilderModel : public CBuilderObject {
public:
  const CBuilderObjectList *GetChildren() const;
  const CBuilderModelList *GetModels() const;

  const CBuilderModelList *GetModels(CString partName) const;
  const CBuilderAtomList *GetAtoms(CString partName) const;
  const CBuilderModelReferenceList *GetModelReferences(
                                   CString refPartName) const;
  const CBuilderAtomReferenceList *GetAtomReferences(
                                   CString refPartName) const;
  const CBuilderConnectionList *GetConnections(CString name) const;
  const CBuilderSetList *GetSets(CString name) const;

  void GetAspectNames(CStringList &list);

  CBuilderModel *CreateNewModel(CString partName);
  CBuilderAtom  *CreateNewAtom(CString partName);
  CBuilderModelReference *CreateNewModelReference(CString refPartName,
                                          CBuilderObject* refTo);
  CBuilderAtomReference *CreateNewAtomReference(CString refPartName,
                                          CBuilderObject* refTo);
  CBuilderSet *CreateNewSet(CString partName);
```

```
  CBuilderConnection *CreateNewConnection(CString connName,
                                          CBuilderObject *src,
                                          CBuilderObject *dst);

  virtual void TraverseModels(void *pointer = 0);
  virtual void TraverseChildren(void *pointer = 0);
};
```

The CBuilderAtom class does not provide any new public methods.

```
class CBuilderAtom : public CBuilderObject {
public:
};
```

The CBuilderAtomReference class provides the GetReferred function that returns the atom (or atom reference) referred to by the given reference.

```
class CBuilderAtomReference : public CBuilderObject {
  const CBuilderObject *GetReferred() const;
};
```

Even though the GME deals with ports of models (since models cannot be connected directly, these are the objects that can be), the component interface avoids using ports for the sake simplicity. However, model references mandate the introduction of a new kind of object, model reference ports. A model reference contains a list of port objects. The GetOwner method of the CBuilderReferencePort class return the model reference containing the given port. The GetAtom method returns the atom that corresponds to the port of the model that the model reference port represents.

```
class CBuilderReferencePort : public CBuilderObject {
public:
  const CBuilderModelReference *GetOwner() const;
  const CBuilderAtom *GetAtom() const;
};
```

The CBuilderModelReference class provides the GetRefered function that returns the model (or model reference) referred to by the given reference. The GetRefereePorts return the list of CBuilderReferencePorts.

```
class CBuilderModelReference : public CBuilderObject {
  const CBuilderReferencePortList &GetRefereePorts() const;
  const CBuilderObject *GetReferred() const;
};
```

A CBuilderConnection instance describes a relation among three objects. The owner is the model that contains the given connection (i.e. the connection is visible in that model). The source (destination) is always an atom or a reference port. If it is an atom then it is either contained by the owner, or it corresponds to a port of a model contained by the owner. So, in case of atoms, either the source (destination) or its parent is a child of the owner. In case of a reference port, its owner must be a child of the owner of the connection.

```
class CBuilderConnection : public CBuilderObject {
public:
  CBuilderModel *GetOwner()  const;
  CBuilderObject *GetSource() const;

  CBuilderObject *GetDestination() const;
};
```

The CBuilderSet class member function provide straightforward access to the different components of sets.

```
class CBuilderSet : public CBuilderObject {
public:
  const CBuilderModel *GetOwner() const;
  const CBuilderObjectList *GetMembers() const;

  bool AddMember(CBuilderObject *part);
  bool RemoveMember(CBuilderObject *part);
};
```

Example

The following simple paradigm independent interpreter displays a message box for each model in the project. For the sake of simplicity, it assumes that there is no folder hierarchy in the given project. The component.cpp file is shown below.

```
#include "stdafx.h"
#include "Component.h"

void CComponent:: InvokeEx(CBuilder &builder,CBuilderObject *focus,
      CBuilderObjectList &selected, long param)
{
  const CBuilderFolderList *folds = builder.GetFolders();
  POSITION fPos = folds->GetHeadPosition();
  while(fPos) {
    CBuilderFolder *fold = folds->GetNext(fPos);
    const CBuilderModelList *roots = fold->GetRootModels();
    POSITION rootPos = roots->GetHeadPosition();
    while(rootPos)
      ScanModels(roots->GetNext(rootPos),fold->GetName());
  }
}

void CComponent::ScanModels(CBuilderModel *model, CString fName)
{
  AfxMessageBox(model->GetName() + " model found in the " +
                fName + " folder");

  const CBuilderModelList *models = model->GetModels();
  POSITION pos = models->GetHeadPosition();
  while(pos)
    ScanModels(models->GetNext(pos),fName);
}
```

Extending the Component Interface

The previous example used the build-in classes only. The component writer can extend the component interface by her own classes. In order for the interface to be able to create the builder object network instantiating the new added classes before the user defined interpretation actually begins, a pair of macros must be used.

The derived class declaration must use one of the DECLARE macros. The implementation must include the appropriate IMPLEMENT macro. There is a pair of macros for models, atoms, model- and atom references, connections and sets. The following list describes their generic form.

```
DECLARE_CUSTOMMODEL(<CLASS>,<BASE CLASS>)
DECLARE_CUSTOMMODELREF(<CLASS>,<BASE CLASS>)
DECLARE_CUSTOMATOM(<CLASS>,<BASE CLASS>)
DECLARE_CUSTOMATOMREF(<CLASS>,<BASE CLASS>)
DECLARE_CUSTOMCONNECTION(<CLASS>,<BASE CLASS>)
DECLARE_CUSTOMSET(<CLASS>,<BASE CLASS>)

IMPLEMENT_CUSTOMMODEL(<CLASS>,<BASE CLASS>,<NAMES>)
IMPLEMENT_CUSTOMMODELREF(<CLASS>,<BASE CLASS>,<NAMES>)
IMPLEMENT_CUSTOMATOM(<CLASS>,<BASE CLASS>,<NAMES>)
IMPLEMENT_CUSTOMATOMREF(<CLASS>,<BASE CLASS>,<NAMES>)
IMPLEMENT_CUSTOMCONNECTION(<CLASS>,<BASE CLASS>,<NAMES>)
IMPLEMENT_CUSTOMSET(<CLASS>,<BASE CLASS>,<NAMES>)
```

Here, the <CLASS> is the name of the new class, while the <BASE_CLASS> is the name of one of the appropriate built-in class or a user-derived class. (The user can create abstract base classes as discussed later.) The <NAMES> argument lists the names of the kinds of models the given class will be associated with. It can be a single name or a comma separated list. The whole names string must be encompassed by double quotes.

For example, if we have a "Compound" model in our paradigm, we can create a builder class for it the following way.

```
// Component.h

class CCompoundBuilder : public CBuilderModel
{
  DECLARE_CUSTOMMODEL(CCompoundBuilder, CBuilderModel)
public:
  virtual void Initialize();
  virtual ~CCompoundBuilder();

// more declarations
};

// Component.cpp

IMPLEMENT_CUSTOMMODEL(CCompoundBuilder, CBuilderModel, "Compound")

void CCompoundBuilder::Initialize()
{
  // code that otherwise would go into a constructor

  CBuilderModel::Initialize();
}

CCompoundBuilder::~CCompoundBuilder()
{
  // the destructor
}
// more code
```

The macros create a constructor and a Create function in order for a factory object to be able to create instances of the given class. Do not define your own constructors, use the Initialize() function instead. You have to call the base class implementation. These macros call the standard MFC DECLARE_DYNCREATE and IMPLEMENT DYNCREATE macros.

If you want to define abstract base classes that are not associated with any of your models, use the appropriate macro pair from the list below. Note that the <NAMES> argument is missing because there is no need for it.

```
DECLARE_CUSTOMMODELBASE(<CLASS>,<BASE CLASS>)
DECLARE_CUSTOMMODELREFBASE(<CLASS>,<BASE CLASS>)
DECLARE_CUSTOMATOMBASE(<CLASS>,<BASE CLASS>)
DECLARE_CUSTOMATOMREFBASE(<CLASS>,<BASE CLASS>)
DECLARE_CUSTOMCONNECTIONBASE(<CLASS>,<BASE CLASS>)
DECLARE_CUSTOMSETBASE(<CLASS>,<BASE CLASS>)

IMPLEMENT_CUSTOMMODELBASE(<CLASS>,<BASE CLASS>)
IMPLEMENT_CUSTOMMODELREFBASE(<CLASS>,<BASE CLASS>)
IMPLEMENT_CUSTOMATOMBASE(<CLASS>,<BASE CLASS>)
IMPLEMENT_CUSTOMATOMREFBASE(<CLASS>,<BASE CLASS>)
IMPLEMENT_CUSTOMCONNECTIONBASE(<CLASS>,<BASE CLASS>)
IMPLEMENT_CUSTOMSETBASE(<CLASS>,<BASE CLASS>)
```

For casting, use the BUILDER_CAST(CLASS, PTR) macro for casting a builder class pointer to its derived custom builder object pointer.

Example

Let's assume that our modeling paradigm has a model kind called Compound. Let's write a component that implements an algorithm similar to the previous example. In this case, we'll scan only the Compound models. Again, the folder hierarchy is not considered. Here is the Component.h file:

```
#ifndef GME_INTERPRETER_H
#define GME_INTERPRETER_H

#include "Builder2000.h"

#define NEW_BON_INVOKE
//#define DEPRECATED_BON_INVOKE_IMPLEMENTED

class CComponent {
public:
  CComponent() : focusfolder(NULL) { ; }
  CBuilderFolder *focusfolder;
  CBuilderFolderList selectedfolders;
  void InvokeEx(CBuilder &builder,CBuilderObject *focus,
       CBuilderObjectList &selected, long param);
};

class CCompoundBuilder : public CBuilderModel
{
  DECLARE_CUSTOMMODEL(CCompoundBuilder, CBuilderModel)
public:
  void Scan(CString foldName);
};

#endif // whole file
```

The component.cpp file is shown below.

```
#include "stdafx.h"
#include "Component.h"

void CComponent::InvokeEx(CBuilder &builder,CBuilderObject *focus,
      CBuilderObjectList &selected, long param)
{
  const CBuilderFolderList *folds = builder.GetFolders();
  POSITION foldPos = folds->GetHeadPosition();
  while(foldPos) {
    CBuilderFolder *fold = folds->GetNext(foldPos);
    const CBuilderModelList *roots = fold->GetRootModels();
    POSITION rootPos = roots->GetHeadPosition();
    while(rootPos) {
      CBuilderModel *root = roots->GetNext(rootPos);
      if(root->IsKindOf(RUNTIME_CLASS(CCompoundBuilder)))
        BUILDER_CAST(CCompoundBuilder,root)->Scan(fold->GetName());
    }
  }
}

IMPLEMENT_CUSTOMMODEL(CCompoundBuilder, CBuilderModel, "Compound")

void CCompoundBuilder::Scan(CString foldName)
{
  AfxMessageBox(GetName() + " model found in " + foldName +
             " folder");

  const CBuilderModelList *models = GetModels("CompoundParts");
  POSITION pos = models->GetHeadPosition();
  while(pos)
    BUILDER_CAST(CCompoundBuilder,models->GetNext(pos))->
                                          Scan(foldName);
}
```

How to create a new component project

To create a new component, run *CreateNewComponent.exe* that comes as part of the GME distribution. A dialog box (Create New Component) is presented to specify the target directory and the component technology to be used. To work with the interface described above, select Builder Object Network.

The second dialog box (Component Configurator) lets you specify the most important characteristics of the component:

- Its type: Interpreter, Plugin or AddOn. (AddOns are not available when using Builder Object Network.)
- The component name
- The name of the paradigm(s) this component is associated with. Multiple paradigms can be specified in a space-separated list.
- The component progID
- The component classname and the component type library name
- The UUID-s associated with the component class and its type library
- The location of the GME 2000 interface files (IDL files) this component is to be compiled to.

The resulting configuration is a ready-to-compile Visual Studio workspace (Component.dsw). If the Builder Object Network is selected, a simple Component.cpp and Component.h files are generated. To user is expected to implement the component by modifying these two files and adding other files if necessary. The other files in the workspace are normally not modified by the user, and for this reason they are generated with read-only attribute.

ConfigureComponent.exe, the application that brings up the Component Configurator dialog box can be run any time to change component attributes. The output is generated to the file specified by the –f command-line argument. It defaults to ComponentConfig.h.

The appendix describes the procedure in detail. After you completed the steps outlined there, you can build the new component dll. This component dll is registered and associated with the paradigms you specify. When you edit a model using one of these paradigms and press the interpret button, you launch this component (if there are more than one components associated with the given paradigm, a menu will pop up to choose from). The dll will be located and loaded at this time.

Appendix A - Database Setup

GME 2000 Database Connectivity

The GME 2000 application provides concurrent access to projects stored in a database. The underlying database access mechanism is based on the Microsoft Repository engine to interface with a SQL database server. Currently only Microsoft SQL Server 7.0 is supported, but other servers, such as Oracle, should work fine. There are four main installation steps:

- Server side installation (creating databases)
- Client side installation Step #1 (GME 2000 and Microsoft Repository)
- Client side installation Step #2 (setting up ODBC data source names)
- Creation of an empty project (only once for each database)

We will not cover the server side installation in detail because it depends on the specific environment and SQL server being used. The SQL server administrator should perform the following steps:

- 1.1. Create an empty dedicated database for each project,
- 1.2. Create (or select) database users,
- 1.3. Give "create" permission(s) to each user.

A copy of GME 2000 should be installed on each client machine along with the Microsoft Repository engine.

- 2.1. Install GME 2000.
- 2.2. Install the Microsoft Repository engine (version 2.1) by executing "msr21.exe" included in the GME 2000 release (also available at http://msdn.microsoft.com/repository/downloads/engine/download.asp)

On each client machine (Windows 95, 98, NT or 2000) the user should set up ODBC data source names. Open Database Connectivity (ODBC) is a Microsoft Corp. defined interface for accessing data from database management systems. The client identifies databases by Data Source Names (DSNs). Each DSN represent an ODBC connection to a specific database by a specific database user.

- 3.1. Open up the "Control Panel"

- 3.2. Find the ODBC Data Sources icon (or similar on other variants of Windows) and double click on it.
- 3.3. Select the System DSN (or User DSN) tab on the dialog box.
- 3.4. Click "Add..." and select SQL server driver.
- 3.5. Now you have to identify the server (ask the SQL server administrator for help). Give the desired name and description to your database connection.
- 3.6. Select the server from the drop down list. If you cannot find it in the list, type in the TCP/IP address, like "server.company.com". Press "Next..."
- 3.7. Select the SQL server authentication. Click on "Client configuration" and select "TCP/IP" (or ask your SQL server administrator), and press OK.
- 3.8. You have to fill in the Login ID and Password fields. These should be the database user name and password, and not the name and password of your local account. Press OK.
- 3.9. If at this point you cannot continue due to some error, ask your SQL server administrator for help.
- 3.10. Set the "default database" to the database containing the project. Proceed through the next few dialog boxes by accepting the default options. Test the data source when given the choice and complete the setup of the data source.

The SQL server administrator should create an empty project for each database. The Microsoft Repository will install the necessary database schema and create an empty project in the database.

- 4.1. Start the GME 2000 application on one of the client machines.
- 4.2. From the "File" menu select "New Project".
- 4.3. Select the paradigm. At this point it is wise to parse the paradigm from an XML file (select "Add from File" and select the XML file). The parser will create a file with extension "MTA". Copy this MTA file to each client machine and register it using "New Project" and "New from File".
- 4.4. Select "Connect to database" and press "Next".
- 4.5. Find the data source name, select it and press "OK".
- 4.6. Type in the password, and press "OK".

Once an empty project is created, users should open up the project from their machine. Here are the steps to access the database.

- 5.1. Start the GME 2000 application.
- 5.2. Obtain the generated "MTA" paradigm file (see point 4.3) and copy it to your machine.
- 5.3. Try out the paradigm file: Select "New Project" from the "File" menu, install the paradigm file by selecting "New from File" and press "OK". Then select "Create project file", and provide a filename. This will create a local project on your machine.

- 5.4. Once the paradigm is installed, you are ready to open up the empty project in the database. Select "Open Project" from the "File" menu.
- 5.5. Select "Connect to database", and locate your DSN. Fill in the password (if required) and press "OK".

Note

Although Microsoft Repository can work on top of Microsoft Access, we do not recommend it because of certain limitations of Microsoft Access. However, for small projects this provides another way to use the multi user environment. Select "New Project" from the "File" menu, select the desired paradigm and then "Create project file". Then from the "Files of type" drop down list select "Microsoft Access Files".

Appendix B - MCL

The Multigraph Constraint Language

The MGA Constraint Language (MCL) is a first-order predicate language used to specify invariant expressions in MGA modeling paradigms. These MCL expressions (i.e. constraints) are contained within constraint blocks in the XML representation of an MGA paradigm. MCL is strongly based on the Unified Modeling Language (UML) Object Constraint Language (OCL). MCL is a subset of OCL, with some MGA-specific extensions designed to make MGA constraint specification easier.

Constraints may be checked on demand or may be event-based. The modeler may check constraints for the currently open model by invoking the "Check" menu option or by pressing the Check button on the toolbar. The modeler may check constraints for all objects in the project by invoking the "Check All" menu option. Event-based constraints provide the additional capability to be checked when the object associated with a constraint receives an event from the GME.

Any number of constraint blocks may be added to an object block. Constraint blocks should be placed immediately following the *dispname* element of the enclosing object block. If the *dispname* element is absent, the constraint blocks should be placed immediately following the opening tag of the enclosing object block. The syntax of a constraint block is as follows:

```
<constraint name="cName" type="cType" eventmask="cEventmask"
depth="cDepth" priority="cPriority">

<![CDATA[ cExpression ]]>

<dispname> descriptionString </dispname>

</constraint>
```

cName: a string containing no spaces used to uniquely identify the constraint

cType: one of the following three values:

eventbased [default] – specifies that the constraint will be evaluated when the object to which it is attached receives one of the events specified in its eventmask.

ondemand – specifies that the constraint will only be evaluated when the modeler either invokes the “Check” command on the associated object or invokes the “Check All” command.

function – specifies that this is a user-defined constraint helper function that should not be evaluated by itself. Refer to the section on functions for further information.

cEventmask: determines which events will cause the evaluation of this constraint. Eventmask values are listed in Table I. If the eventmask attribute is not specified, it is given a default value of zero.

Name	Value	Description
OBJEVENT_CREATED	0x80000000	The object has been created
OBJEVENT_DESTROYED	0x40000000	The object has been destroyed
OBJEVENT_ATTR	0x00000001	Attribute changed
OBJEVENT_REGISTRY	0x00000002	Registry changed
OBJEVENT_NEWCHILD	0x00000004	Child added
OBJEVENT_RELATION	0x00000008	Reference pointer, set member, connection endpoint changed
OBJEVENT_PROPERTIES	0x00000010	Name, etc. changed
OBJEVENT_PARENT	0x00000100	Object has been moved
OBJEVENT_LOSTCHILD	0x00000200	Child removed/moved away
OBJEVENT_REFERENCED	0x00000400	Object has been referenced
OBJEVENT_CONNECTED	0x00000800	Object has been connected
OBJEVENT_SETINCLUDED	0x00001000	Object has been included in set
OBJEVENT_REFRELEASED	0x00002000	Object reference has been released
OBJEVENT_DISCONNECTED	0x00004000	Object has been disconnected
OBJEVENT_SETEXCLUDED	0x00008000	Object has been excluded from set

Table I

cPriority: determines the order of evaluation among the constraints attached to an object. Allowable values are the integers between 1 (highest) and 10 (lowest). In addition, the violation of a constraint with a priority of 1 as a result of an event-based evaluation will prevent the current transaction from being committed. The default priority value is 5.

cDepth: one of the following three values:

0 – evaluate the constraint when its associated object receives an event specified by the eventmask parameter

1 [default] – evaluate the constraint when its associated object or any of its immediate children receives an event specified by the eventmask parameter

any – evaluate the constraint when its associated object or any of its descendants receives an event specified by the eventmask parameter

cExpression: an MCL first-order expression involving the names and types of various modeling objects, along with various MCL operators listed in the next section.

descriptionString: a string describing the constraint in natural language. When the constraint manager is used in interactive mode, this string will appear as part of the error message when a constraint violation occurs.

MCL Operators

Arithmetic operators: +, -, *, /, =, <, >, <=, >=, <>

Logical operators: and, or, xor, not, implies, if, then, else, endif

Collection operator: ->

Property operator: .

Functional operators:

where
- c : collection
- str : string
- fco : first-class object
- fdr : folder
- r : reference fco
- s : set fco
- cpt : connection point

<u>Operations on objects:</u>

{fco,fdr}.name() **: string**

Returns a string representing the name of the caller.

{fco,fdr}.parent() **: fco**
: fdr

Returns the parent object of the caller. If caller has no parent, returns null.

{fco,fdr}.kindName() **: string**

Returns a string representing the kind name of the caller.

fco.roleName() **: string**

Returns a string representing the role name of the caller.

{fco,cpt,fdr}.isNull() **: boolean**

Returns true if caller is null, false otherwise.

str.intValue() **: int**

Returns the result of parsing the caller as a string representation of an integer.

str.doubleValue() **: double**

Returns the result of parsing the caller as a string representation of an double.

fco.connected(| roleName:string) **: collection of fco's**

Returns a collection of fco’s to which the caller is connected via a connection object.

roleName: filters the collection of fco’s by the role of the *target fco*

fco.connectedAs(| roleName:string) **: collection of fco’s**

Returns a collection of fco’s to which the caller is connected via a connection object.

roleName: filters the collection of fco’s by the role of the *caller* in each connection (i.e., only the cpt’s of the caller that have the role of roleName will be examined)

fco.connections(| kindName:string) : collection of cpt's

Returns the collection of connection points belonging to the caller.

kindName: filters the collection of cpt’s by the kind of their owning connection

fco.connectionPoints(| connRole:string) : collection of cpt's

Returns the collection of connection points belonging to the caller.

connRole: filters the collection of cpt’s by the role of their owning connection

r.usedByConnections(| kindName:string) : collection of cpt's

Returns the collection of connection points that use the calling reference object.

kindName: filters the collection of cpt’s by the kind of their owning connection

s.members() : collection of fco's

Returns the collection of fco’s that are members of the calling set object.

fco.refersTo() : fco

Returns the fco referred to by caller.

fco.referencedBy() : collection of fco's

Returns the collection of fco’s that reference the caller.

fco.models(| kindName:string) : collection of fco's

Returns all descendant models of the caller (recursive search).

kindName: filters collection of fco’s by kind

fco.atoms(| kindName:string) : collection of fco's

Returns all descendant atoms of the caller (recursive search).

kindName: filters collection of fco’s by kind

fco.parts(| roleName:string) : collection of fco's

Returns all child parts (fco’s) of the caller.

roleName: filters collection of fco’s by role

fco.modelParts(| roleName:string) : collection of fco's

Returns all child models of the caller.

roleName: filters collection of fco’s by role

fco.atomParts(| roleName:string) : collection of fco's

Returns all child atoms of the caller.

roleName: filters collection of fco’s by role

fco.referenceParts(| roleName:string) : collection of fco's

Returns all child references of the caller.

roleName: filters collection of fco’s by role

fco.connectionParts(| roleName:string) : collection of fco's

Returns all child connections of the caller.

roleName: filters collection of fco’s by role

fco.setParts(| roleName:string) : collection of fco's

Returns all child sets of the caller.

roleName: filters collection of fco’s by role

fco.memberOfSets(| kindName:string) : collection of fco's

Returns the collection of sets that include the caller as a member.

roleName: filters collection of fco’s by kind

fco.subTypes() : collection of fco's

Returns all fco’s that are subtypes of the caller.

fco.instances() : collection of fco's

Returns all fco’s that are instances of the caller.

fco.type() : fco

Returns the parent fco of the caller in the type hierarchy.

fco.baseType() : fco

Returns the base fco of the caller in the type hierarchy.

fco.isType() : boolean

Returns true if the caller is a type, false if the caller is an instance.

fco.isInstance() : boolean

Returns true if the caller is an instance, false if the caller is a type.

fco.folder() : fdr

Returns the folder that contains the caller in the object hierarchy.

fco.attribute(attributeName : string) : string

: int

: boolean

Returns the value of the attribute named attributeName associated with the calling object.

cpt.role() : string

Returns a string representation of the role of the calling connection point.

cpt.target() : fco

Returns the fco with which the calling connection point is associated.

cpt.references() : collection of fco's

Returns the collection of fco’s that reference the caller.

cpt.owner() : fco

Returns the owning connection of the calling connection point.

fdr.folders() : collection of fdr's

Returns all descendant folders of the calling folder (recursive).

fdr.childFolders() : collection of fdr's

Returns all child folders of the calling folder.

fdr.rootDescendants() **: collection of fco's**

Returns all root fco's that are descendants of the calling folder (recursive).

fdr.rootChildren() **: collection of fco's**

Returns all root fco's that are children of the calling folder.

fdr.allDescendants() **: collection of fco's**

Returns all fco's that are descendants of the calling folder (recursive).

<u>Operations on collections of objects:</u>

c->size() **: integer**

Returns the number of objects in the collection.

c->forAll(x | f(x)) **: Boolean**

If f(x) is true for all x in c, returns true; else, returns false.

c->exists(x | f(x)) **: Boolean**

If f(x) is true for any x in c, returns true; else, returns false.

c->select(x | f(x)) **: collection**

Returns a collection containing all x in c for which f(x) is true.

c->includes(inclObject:fco

| inclCpt:cpt | inclFdr:fdr) **: Boolean**

Returns true if the collection contains the parameter, false otherwise.

c->union(c2:collection) **: collection**

Returns a collection containing the union of the objects in c and c2.

c->intersection(c2:collection) **: collection**

Returns a collection containing the intersection of the objects in c and c2.

c->theOnly() **: fco**

: cpt

: fdr

If c contains a single object, returns the object. Otherwise, returns false.

Functions

The modeler may define his/her own functional operators using the built-in operators defined in this document. Functions are enclosed in constraint blocks in the XML representation of the modeling paradigm, but the syntax for denoting these user-defined functions differs slightly from the syntax for constraints, as shown below. Function constraint blocks are placed in the root folder of the project and are accessible to all constraints within the project.

```
<constraint type="function">
   <![CDATA[ function funcName ( {parName: parType}* ) fExpression ]]>
</constraint>
```

funcName: a string containing no spaces used to uniquely identify the function

*{parName : parType}** : the formal parameter list of the function, consisting of any number of parameter name and type pairs. The following parTypes are allowed: Integer, Double, String, Boolean, Object, ObjectList, ConnPoint, ConnPointList, Folder, and FolderList.

fExpression: same as *cExpression* for a constraint.

Examples

The following constraint will be violated if any of the descendant models of the object to which the constraint is attached have the same name. This constraint has a priority of 2 will be evaluated whenever the object or any of its immediate children receive a properties change event or a new child event.

```
<constraint name="UniqueDescendantNames" type="eventbased"
eventmask="0x00000014" depth="1" priority="2">
<![CDATA[models()->forAll(m1, m2 | m1.name = m2.name implies m1 =
m2)]]>
<dispname>No descendant models may have the same name.</dispname>
</constraint>
```

The following function returns true if the calling object contains N atoms having the role of Parameters:

```
<constraint type="function">
   <![CDATA[function containsNParameterAtoms(N : Integer)
atomParts("Parameters")->size = N ]]>
</constraint>
```

Appendix C – References

Model Integrated Computing References

The following references provide detailed information on Model Integrated Computing technology, development, and application:

S. White, et al.: “Systems Engineering of Computer-Based Systems", IEEE Computer, pp. 54-65, November 1993.

J. Sztipanovits, et al.: “MULTIGRAPH: An Architecture for Model-Integrated Computing,” Proceedings of the IEEE ICECCS’95, pp. 361-368, Nov. 1995.

D. Oliver, T. Kelliher, J. Keegan, Jr., Engineering Complex Systems with Models and Objects. New York: McGraw-Hill, 1997.

J. Sztipanovits, “Engineering of Computer-Based Systems: An Emerging Discipline,” Proceedings of the IEEE ECBS’98 Conference, 1998.

Ledeczi A., Maroti M., Bakay A., Karsai G., Garrett J., Thomason IV C., Nordstrom G., Sprinkle J., Volgyesi P.: The Generic Modeling Environment, Workshop on Intelligent Signal Processing, accepted, Budapest, Hungary, May 17, 2001

Ledeczi A., Nordstrom G., Karsai G., Volgyesi P., Maroti M.: On Metamodel Composition, IEEE CCA 2001, CD-Rom, Mexico City, Mexico, September 5, 2001

Ledeczi, et al., “Metaprogrammable Toolkit for Model-Integrated Computing," Proceedings of the IEEE ECBS’99 Conference, 1999.

Additionally, many other MIC-related journal articles, conferences papers, and other reference materials are available from the ISIS web site, accessible via the following URL:

http://www.isis.vanderbilt.edu/

Glossary of Terms

aspects

The parts contained within a GME model are partitioned into viewable groups called aspects. Parts may be added or deleted only from their primary aspects, but may be visible in many secondary aspects.

CBS

Computer Based System

Compound model

A model that can contain other objects

connection

A line with a particular appearance and directionality joining two atomic parts or parts contained in models. In the GME, connections can have domain-specific attributes (accessed by right-clicking anywhere on the connection).

CORBA

Common Object Request Broker Architecture

COTS

Commercial off-the-shelf software

DSME

Domain Specific MIPS Environment

GME

See Generic Model Environment

GOTS

Government off-the-shelf software

Generic Modeling Environment

A configurable, multi-aspect, graphical modeling environment used in the MultiGraph Architecture

interpreters

See Model interpreters

Link

See Link parts

Link parts

Atomic parts contained within a model that are visible, and can participate in connections, when the container model appears inside other models.

MCL

MGA constraint language. A subset of OCL, with MGA-specific additions.

Metamodel

A model that contains the specifications of a domain-specific MIPS environment (DSME). Metamodels contain syntactic, semantic, and presentation specifications of the target DSME.

metamodeling environment

A domain-specific MIPS environment (DSME) configured to allow the specification and synthesis of other DSMEs.

MGA

See MultiGraph Architecture

MGK

MultiGraph Kernel. Middleware designed to support real-time MultiGraph execution environments

MIC

Model Integrated Computing

MIPS

Model Integrated Program Synthesis

Model interpreters

High-level code associated with a given modeling paradigm, used to translate information found in the graphical models into forms (executable code, data streams, etc.) useful in the domain being modeled.

Model translators

See Model interpreters

modeling paradigm

The syntactic, semantic, and presentation information necessary to create models of systems within a particular domain.

MultiGraph Architecture

A toolset for creating domain-specific modeling environments.

OCL

Object Constraint Language (a companion language to the UML)

paradigm

See modeling paradigm

POSIX

Portable Operating System Interface, An IEEE standard designed to facilitate application portability

Primitive model

A model that cannot contain other models

Reference parts

Objects that refer to (i.e. *point to*) other objects (atomic parts or models)

References

See Reference parts